Содержание

1. Lua за 15 минут **2**
1.1. Комментарии 2
1.2. Переменные (простые типы) 2
1.3. Логические операторы 3
1.4. Операторы отношений 4
1.5. Условный оператор if 4
1.6. Циклы 6
1.6.1 While 6
1.6.2 Repeat 6
1.6.3 Числовой for 7
1.6.4 Общий for 8
1.7. Присваивание 8
1.8. Блоки. Глобальные и локальные переменные 9
1.9. Таблицы 10
1.9.1 Общая информация 10
1.9.2 Итерация элементов таблицы 12
1.9.3 Массивы 13
1.9.4 Множества (Sets) 14
1.9.5 Функции для работы с таблицами 15
1.10. Работа со строками 17
1.10.1 Шаблоны 21
1.11. Функции 23
1.11.1 Множественные значения 24
1.11.2 Переменное число параметров. 25
1.11.3 Именованные аргументы. 26
1.11.4 Особенности. 26

1. Lua за 15 минут

Данное пособие является адаптацией статьи “Learn Lua in 15 Minutes” с некоторыми дополнениями. Оригинал на английском языке можно найти по адресу: http://tylerneylon.com/a/learn-lua/.

1.1. Комментарии

Комментарии в Lua можно сделать двумя способами:

```
1 -- One line comment
2
3 --[[
4     first line
5     seconde line
6 --]]
```

Первый способ начинает однострочный комментарий, второй — многострочный.

1.2. Переменные (простые типы)

Все числовые переменные являются вещественными (double):

```
1 number = 42
2 another_number = 3.1415
```

Над числами можно проводить следующие операции: сложение (+), вычитание (-), умножение (*), деление (/), возведение в степень (^):

```
1 add = 5 + 3 -- add = 8
2 sub = add - 4 -- sub = 4
3 mult = add * sub -- mult = 32
4 div = add / sub -- div = 2
5 pow = 2^3 -- pow = 8
```

Строки в языка Lua являются *неизменяемыми*, то есть нельзя обратиться к индексу строки и поменять символ. Объявление сток можно сделать тремя способами:

```
1 color = 'black'
2 season = "summer"
3 huge_string = [[ This is
```

```
4           a very-very
5           long string! ]]
```

Для соединения строк (*конкатенация* строк) используется оператор ..:

```
1 name = "Petr"
2 surname = "Ivanov"
3 pupil = name .. " " .. surname -- pupil = "Petr Ivanov"
```

Если при конкатенации строк будут использоваться числовые переменные, то они автоматически будут приведены к строкам:

```
1 number = 42
2 question = number .. " is good answer for everything!"
3 -- question = "42 is good answer for everything!"
```

Переменные могут принимать логическое значение *boolean*: **true** (истина) или **ложь**:

```
1 to_be_or_not_to_be = true
```

Переменные также могут принимать значение *nil*. Данный тип означает, что значения у переменной **не существует**!

```
1 aliens_exist = nil
```

1.3. Логические операторы

Существуют следующие логические операторы: `and`, `or` и `not`. Все логические операторы предполагают, что `false` и `nil` представляют собой значение **false**, а все остальные значения — **true**.

Оператор `and` возвращает первый аргумент в том случае, если его значение *false*, в противном случае возвращается второй аргумент. Оператор `or` возвращает первый аргумент в том случае, если его значение *true*, в противном случае возвращается второй аргумент.

```
1 print(4 and 5)          -- 5
2 print(nil and 13)       -- nil
3 print(false and 13)     -- false
4 print(4 or 5)           -- 4
5 print(false or 5)       -- 5
```

Операторы `and` и `or` не вычисляют второй аргумент, если в это нет необходимости. Например, выражение `x = x or v` эквивалентно следующему выражению:

```
1 if not x then x = v end
```

То есть, если значение `x` не существует, то ставится значение `v`.

Ещё один вариант использования условных операторов: реализация тернарного оператора (`a ? b : c`). В языке Lua его можно реализовать следующим способом:

```
1 a and b or c -- (a and b) or c
```

Пример выбора максимального значения из двух чисел:

```
1 max = (x > y) and x or y
```

Сперва вычисляется выражение `x > y`. Если оно имеет значение `true`, то срабатывает `(x > y) and x` и возвращается `x`, так как `x` — число и всегда равен значению `true`. Если же выражение `x > y` имеет значение `false`, то выражение `(x > y) and x` возвращает `false`, оно сравнивается с `y`, и оператор `or` возвращает значение `y`.

Оператор `not` всегда возвращает `true` или `false`:

```
1 print(not nil)      -- true
2 print(not false)    -- true
3 print(not 0)        -- false
4 print(not not nil)  -- false
```

1.4. Операторы отношений

В языке Lua выделяются следующие операторы отношений, каждый из которых возвращает `true` или `false`:

```
1 <   >   <=   >=   ==   ~=
```

Оператор == проверяет равенство аргументов, а оператор `~=` — неравенство:

```
1 print(5 == 6) -- false
2 print(52 ~= 0) -- true
```

1.5. Условный оператор if

Условия в языке Lua записываются при помощи условного оператора `if`:

```
1 if statement then
2 ... -- do something if statement == true
3 end
```

Оператор проверяет условие *statement* и выполняет операции между ключевыми словами `then` и `end` только в том случае, если *statement* — истинен.

Примеры условий:

```
1 if a < 0 then a = 0 end
2
3 if object == "car" then
4   print("This is car!")
5 end
```

Можно задавать поведение условного оператора `if` при помощи ключевого слова `else`, в случае, если условие *statement* — ложно:

```
1 if statement then
2 ... -- statement == true
3 else
4 ... -- statement == false
5 end
```

Пример использования:

```
1 if age < 18 then
2   print("You can't go to this movie!")
3 else
4   print("Your age is allowed for this movie")
5 end
```

Иногда могут понадобится для работы множественные ветвления (`elseif`) условного оператора `if`:

```
1 if op == "+" then
2   r = a + b
3 elseif op == "-" then
4   r = a - b
5 elseif op == "*" then
6   r = a*b
7 elseif op == "/" then
8   r = a/b
9 else
10  print("Error!")
11 end
```

Отрицание логического выражения *statement* задается при помощи ключевого слова `not`:

```
1 if not end_of_game then ... end
```

Выражение *statement* может содержать в себе сложные логические выражения:

```
1 if age >= 14 and age <= 18 then ... end
```

1.6. Циклы

Циклы — это управляющая конструкция, которая позволяет многократно исполнять ряд инструкций.

1.6.1 While

Цикл с предусловием (`while`) — это цикл, который будет выполняться, пока истинно условие (`true`). То есть если условие истинно, цикл выполняется, иначе он заканчивает свою работу и управление передается коду за ним.

```
1 num = 0
2 while num < 3 do
3   num = num + 1;
4   print(num);
5 end
```

В результате будет выведено:

1

2

3

1.6.2 Repeat

Цикл с предусловием (`repeat`) — цикл, который так же будет выполняться, пока условие истинно (`true`), но проверка условия выполняется после прохождеия тела цикла. То есть тело цикла всегда будет выполняться хотя бы один раз, в отличие от цикла while, который может вообще не выполниться.

```
1 num = 3
2 repeat
3   print(num)
4   num = num - 1
5 until num == 0
```

В результате будет выведено:

3

2

1

1.6.3 Числовой for

Счетный цикл или цикл со счетчиком (`for`) — цикл, в котором некоторая заданная переменная меняет свое значение от заданного начального значения до заданного конечного в соответсвии с указанным шагом.

Синтаксис счетного цикла:

```
1 for var=exp1,exp2,exp3 do
2   something
3 end
```

Действие *something* будет исполняться для каждого значения управляющей переменной *var* от начального значения *exp1* до конечного значения *exp2* с шагом *exp3*. Указывать шаг **необязательно**, так как по умолчанию шаг равен 1.

```
1 for var=0,6,2 do
2   print(var)
3 end
```

В результате будет выведено:

0

2

4

6

Замечания:

- Управляющая переменная *var* является локальной, то есть видна только в пределах цикла, а в не его не существует.
- Если в качестве одного из *exp* стоит функция, то она будет вызвана всего один раз перед началом цикла, то есть при изменении значения переменных, передаваемых в цикл, граница цикла все равно не изменится.
- Не следует менять значение управляющей переменной, так как тогда поведение будет непредсказуемым. Если есть необходимость остановить цикл, лучше использовать оператор `break`.

```
1 var = 3
2 for i = 1,10 do
3   if i >= var then
4     break
5   else
6     print(i .. "is less than 3")
7   end
8 end
```

В результате будет выведено:
1 is less than 3
2 is less than 3

1.6.4 Общий for

Совместный цикл или цикл с итератором(for) — цикл, который позволяет обходить все значения, которые возвращаются функцией итератора. Итератор предоставляет нам доступ к элементам коллекции(массива) и обеспечивает навигацию по ней. Говоря простым языком, совместный цикл позволяет нам "пройтись"по всем элементам массива или другого объединения, последовательно получая индексы и/или значения.

```
1 for i,v in ipairs(a) do
2   print(v)
3 end
```

За один шаг цикла в i помещается очередной индекс массива a, а в v значение, ассоциируемое с данным индексом.

Стандартные функции-итераторы:

- io.lines - обход строк в файле
- pairs - пар в массиве(таблице)
- string.gfind - слов в строке
- и т.д.

1.7. Присваивание

Присваивание означает изменение(запись) значения переменной или поля таблицы(массива) — смотри секцию 1.9.

```
1 str = "Hello" .. "World"
2 number = number + 1
```

Lua позволяет проивзодить множественное присваивание, то есть список значений присваивается списку переменных за один шаг. Элементы обоиз списков разделяются запятыми.

```
1 str, number = "Hello" .. "World", number + 1
```

Переменной *str* будет присвоено значение *Hello World*, в то время как значение переменной *number* увеличится на единицу.

В Lua сначала производится оценка переменных, а затем выполняется присваивание. Благодаря этому, мы можем произвести обмен значениями переменых за один шаг

```
1   x, y = y, x                    -- swap x and y
2   a[i], a[j] = a[j], a[i]        -- swap a[i] and a[j]
```

Если количество переменных отлично от количества значений, то:

1. Если количество переменных больше количества значений, то переменым, которым не представлены значения получат значение *nil*

2. Если количество переменных меньше количества значений, то лишние значения будут проигнорированы

```
1       a, b, c =  1, 2
2       print(a,b,c)             --> 1   2   nil
3       a, b = 1, 2, 3           -- 3 ignores
4       print(a,b)               --> 1   2
5       a, b, c = "Hello"
6       print(a,b,c)             --> "Hello"   nil   nil
```

1.8. Блоки. Глобальные и локальные переменные

Блок является логически сгруппированным набором идущих подряд инструкций и ограничивает область видимости переменных. Блоком является любая управляющая конструкция(циклы, условные конструкции после `then` и `else`, функции), а также файл, в котором исполняется скрипт. Блоки могут быть вложенными. Для создания блока используется конструкция `do-end`.

```lua
do
  --some instructions
end
```

```lua
str = "Hello World"      -- global
local x = 10                    -- local
```

Все переменные объявленые в lua в любом блоке являются глобальными, если они не помечены служебным словом `local`. `local` указывает на то, что переменная будет локальна, то есть сущесвтует только в пределах блока, в котором она объявлена.

```lua
x = 10
i = 1

while i <= x do
  local x = i*2      -- local for "while" body
  print(x)           --> 2, 4, 6, 8, ...
  i = i + 1
end

if i > 20 then
  local x            -- local for "then" body
  x = 20
  print(x + 2)     --> 22
else
  print(x)          --> 10 (global)
end

print(x)              --> 10 (global)
```

1.9. Таблицы

1.9.1 Общая информация

Таблица представляет собой ассоциативный массив. Ассоциативный массив — это массив, в котором индексы не обязательно должны являться числами, а могут быть представлены и другими типами(кроме `nil`), то есть это множество пар "ключ-значение". У таблицы нет фиксированного размера, она может динамически увеличиваться. Таблица может хранить значения различных типов. Если значение поля не задано, то оно будет равно `nil`.

Таблицы используются для представления обычных массивов, очередей, множеств и других структур данных. В Lua таблицы являются объектами. Объект — это некоторая сущность, обладающая определенным состоянием и поведением, имеющая заданные свойства и методы, которые производят операции над ними. При работе с объектами, на самом деле мы манипулируем ссылками на них в памяти. Таблицы объявляются при помощи специального выражения конструктора — {}.

```
1 table = {}
2 table[1] = 4
3 table[2] = "Hello"
4 table["some"] = 1
5 print(table[1] + table["some"])      --> 5
6 print(table[2] .." " .. table[1])    --> Hello 4
7 print(table["other"])                --> nil
```

Работая с таблицей, мы обращаемся к указателю(ссылке) на нее в виртуальном адресном пространстве. В следующем ниже примере переменная b будет указывать на ту же самую таблицу, что и a.

```
1 a = {}
2 a["x"] = 10
3 b = a  -- "b" refers to the same table as "a"
4
5 print(b["x"]) --> 10
6 b[2] = 5
7 print(a[2]) --> 5
8
9 a = nil -- "a" no longer refers to the table
10 b = nil -- "b" also doesn't refer to the table, there isn't any
     reference to the table
```

Таблица существует в памяти до тех пор, пока на нее указывает хоть одна переменная, как только закончатся все ссылки на таблцу, она будет удалена, и память будут освобождена.

Lua допускает другую форму обращения к элементу таблицы, индекс которой является строкой.

```
1 a.x = 10 -- same as a["x"] = 10
```

Стоит отметить, что $a.x$ означает именно обращение к индексу $"x"$, но никак не x.

Если индекс содержит специальные символы, то обращаться к индексу через оператор []:

```
1 a = {}
2 a["this-is-my-special-key"] = "value";
```

Таблицу можно инициализировать при помощи ключей:

```
1 a = {
2   name = "Petr",
3   surname = "Ivanov"
4 }
5
6 print(a.name) -- Petr
7 print(a.surname) -- Ivanov
```

Размер таблицы можно узнать при помощи оператора #:

```
1 my_table = {10, 22, 35, 47}
2 print(#my_table) --> 4
```

1.9.2 Итерация элементов таблицы

Итерацию элементов (перебор всех элементов) можно осуществить при помощи конструкции `for .. in pairs`:

```
1 a = {
2   ["Blondie"] = "Good",
3   ["Angel Eyes"] = "Bad",
4   ["Tuco"] = "Ugly"
5 }
6
7 for i,v in pairs(a) do print("Person " .. i .. " is " .. v) end
```

Вывод на экран:

```
1   Person Tuco is Ugly
2   Person Blondie is Good
3   Person Angel Eyes is Bad
```

В данном примере в цикле `for` в переменную i помещаются ключи таблицы, а в v — значения. Ключи могут быть как и числами, так и строками. Необходимо помнить, что порядок обхода элементов при использовании `for .. in pairs` является *случайным*!

Если все ключи в таблице являются числами, то можно использовать конструкцию `for .. in ipairs`:

```
1 t = { "a", "b", "c", "d" }
2 for i, v in ipairs(t) do
3   print(i .. ": " .. v)
4 end
```

Результат выполнения:

```
1   1: a
2   2: b
3   3: c
4   4: d
```

Отличие от `for .. in pairs` в том, что ключ является целочисленным, начинается с 1, и с каждой итерацией увеличивается на 1. Если во время обхода таблицы будет встречен ключ-строка, то он будет проигнорирован, если же попадет ключ, который равен `nil`, то обход массива будет прекращен!

```
1 t = { "a", ["test"] = "f", "b", "c", ["name"] = 5, "d", nil, 6 }
2 for i, v in ipairs(t) do
3   print(i .. ": " .. v)
4 end
```

Результат выполнения:

```
1   1: a
2   2: b
3   3: c
4   4: d
```

1.9.3 Массивы

Массивы можно реализовать при помощи таблиц, используя в качестве ключей числа. У массивов нет ограничений по количеству элементов, и они “растут” когда это понадобится.

Пример инициализации массивов:

```
1 a = {}    -- new array
2 for i=1, 1000 do
3   a[i] = i + 2
4 end
```

Массивы можно инициализировать также таким способом:

```
1 squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}
```

Необходимо обратить внимание, что индексация массивов в языке Lua начинаются с **1**, а не с 0! Стандартные функции по работе с таблицами подразумевают, что массивы индексируются с 1. Однако, при инициализации можно задать другие границы массива:

```
1 a = {}
2 for i=-5, 5 do
3   a[i] = 0
4 end
```

Двумерные массивы можно реализовать двумя способами:

- Создать массив из N элементов. Каждый элемент массива - это ещё один массив размера M:

```
1 mt = {}          -- create the matrix
2 for i=1,N do
3   mt[i] = {}     -- create a new row
4   for j=1,M do
5     mt[i][j] = 0
6   end
7 end
```

- Создать *одномерный* массив из $N * M$ элементов и добавлять элементы по следующему правилу (смотри 4-ую строку):

```
1 mt = {}          -- create the matrix
2 for i=1,N do
3   for j=1,M do
4     mt[i*M + j] = 0
5   end
6 end
```

Такими же способами можно создавать N-мерные массивы.

1.9.4 Множества (Sets)

При помощи таблиц в Lua можно реализовать множества, если в качестве ключа передавать необходимое значение (число или строку), а в качестве значения — `true`. Множество — это структура данных, которая позволяет хранить ограниченное число значений определенного типа без определенного порядка, при этом

каждое значение встречаеися ровно один раз. В программировании множество конечно. Проверять наличие элемента можно такой конструкцией: `if set[element] then ... end`, где *set* — множество, а *element* — искомый элемент.

Пример множества:

```
1  set = {}
2
3  for i = 2, 10, 2 do
4    set[i] = true
5  end
6
7  for i = 1, 10 do
8    if set[i] then
9      print(i .. " in set")
10   end
11 end
```

1.9.5 Функции для работы с таблицами

1. `table.insert` — производит вставку элемента в таблицу, представляющую собой массив. В качестве аргументов предается таблица, куда будет производиться вставка, затем, может идти индекс позиции для вставки, в этом случае все остальные элементы будут сдвинуты, а затем собственно значение для вставки.

   ```
   1  a = {10, 20, 30}
   2  table.insert(a, 1, 0)      -- a will be: 0, 10, 20, 30
   ```

   Если индекс для вставки не указан, то элемент будет добавлен в конец.

2. `table.remove` — удаляет элемент с указанной позиции. Аргументами является имя таблицы и индекс элемента для удаления. Если индекс не указан, то будет удален последний элемент массива.

   ```
   1  a = {10, 20, 30}
   2  table.remove(a, 2)      -- a will be: 10, 30
   ```

3. `table.concat` — соединяет элементы массива в одну строку. Каждый элемент должен быть конвертируемым в строку. Можно задать разделитель, который будет ставится между элементами при объединении. Так де можно задать диапазон конкатенации.

```
1 table.concat({ 1, 2, "three", 4, "five" }) -- 12three4five
2 table.concat({ 1, 2, "three", 4, "five" }, ", ") -- 1, 2,
    three, 4, five
3 table.concat({ 1, 2, "three", 4, "five" }, ", ", 2) -- 2,
    three, 4, five
4 table.concat({ 1, 2, "three", 4, "five" }, ", ", 2, 4) --
    2, three, 4
```

4. `table.sort` — сортирует элементы массива. В качестве аргументов принимает массив, который необходимо сортировать, и, что является необязательным, функцию для сравнения двух элементов, принимающую 2 параметра, и в случае, если первый должен идти первее в отсортированном массиве, она должна возвращать *true*.

   Пример без функции:

   ```
   1 array={4, 2, 1, 3, 5, 0}
   2 table.sort(array) -- array would be 0, 1, 2, 3, 4, 5
   ```

   Пример с функцией:

   ```
   1 array = {4, 2, 1, 3, 5, 0}
   2 table.sort(array, function(a,b) return a>b end) -- array
       would be 5, 4, 3, 2, 1, 0
   ```

5. `table.foreach` — выполняет для каждого элемента таблицы заданную функцию. На каждой итерации функции передается пара *ключ-значение*, соответствующая элементу текущей итерации.

   ```
   1 array = { 1, 2, "three"; number="four", next="five" }
   2 table.foreach(array, print)
   ```

   Результат выполнения:
   1 1
   2 2
   3 three
   number four
   next five

   Пример с собственной функцией:

   ```
   1 table.foreach({1,"two",3,"four"}, function(k,v)
       print(string.rep(v,k)) end)
   ```

Результат выполнения:
1
twotwo
333
fourfourfourfour

6. `table.foreachi` — аналогична foreachi, только в функцию подается на пара *ключ-значение*, а пара *индекс-значение*, то есть на каждой итерации берется только элемент, ключ которого является числом, то есть элемент, имеющий индекс.

```
1 array = { 1, 2, "three"; number="four", next="five" }
2 table.foreach(array, print)
```

Результат выполнения:
1 1
2 2
3 three

Замечание: `table.foreach` и `table.foreachi` являются устаревшими в версии *lua* 5.1.(depracated). Их использование является нежелательным, так как скорее всего в дальнейших версиях эта вохможность языка будет убрана.

1.10. Работа со строками

Интерпретатор *Lua* имеет достаточно ограниченные возможности по работе со строками. Он может создавать строки и объединять их, но извлечение подстроки, вычисление размера и т.д. представляется для него невозможным. Поэтому для удобной работы со строками в *Lua* используется его стандартная библиотека для строк. Она включает множество различных функций для работы со строками.

1. `string.len` — возвращает длину строки.

```
1 s = "Hello"
2 print(string.len(s))        --> 5
```

2. `string.rep` — возвращает строку, в которой заданная строка повторена n раз.

```
1 s = "Hello"
2 print(string.rep(s, 4))        --> HelloHelloHelloHello
```

3. `string.lower` и `string.upper` — возвращают строку в нижнем или верхнем регистре, соотвественно.

```
1 s = "HeLlo"
2 print(string.lower(s))      --> hello
3 print(string.upper(s))      --> HELLO
```

4. `string.sub` — возвращает подстроку, начиная с позиции i и заканчивая позицией j.Позиция может быть как положительным числом(отсчет с начала слова), так и отрицательным(отсчет с конца). В *Lua* первый символ строки имеет индекс 1 при отсчете с начала строки, при отсчете с конца строки последний символ имеет индекс -1.

```
1 s = "Some string here"
2 t = string.sub(s, 2, -2);
3 print(t);                          -->ome string her
```

Замечание: Функция `string.sub` не именяет саму строку, а создает новую, поэтому для получения подстроки необходимо использовать переменную, в которую будет записана эта подстрока.

5. `string.char` и `string.byte` — функции, которые используются для перевода числа в символьный эквивалент и наоборот. Функция `char` принимает 0 или более целых чисел, переводит каждое в символ, коду которого данное число соотвествует и возвращает строку, которая является совокупностью всех получившихся символов.

```
1 print(string.char(97, 98, 99, 100))       --> abcd
```

Функция `string.byte` возвращает код одного символа, стоящего на i позиции. Если второй аргумент, обозначающий позиция не задан, то будет возвращен код первого символа.

```
1 print(string.byte("abc"))                 -->  97
2 print(string.byte("abc", 2))              -->  98
3 print(string.byte("abc", -1))             -->  99
```

6. `string.format` — функция, которая позвляет отформатировать строку, то есть создать строку заданного формата. Первым аргументом ф функцию передается стркоа формата, а затем данные, которые должны передаваться в эту строку. Строка формата состоит из обыного текста и директив(спецификаторов,

модификаторов), которая контролирует то, где и как каждый следующий поданный аргумент должен располагаться в итоговой строке. Директива начинается с символа

Спецификаторы типа:

(a) `%d` — целое десятичное число со знаком

(b) `%u` — целое десятичное число без знаком

(c) `%f` — число с плавающей запятой в виде десятичной дроби

(d) `%o` — целое восьмеричное число

(e) `%x` — целое шестнадцатеричное число в нижнем регистре

(f) `%s` — строка

Спецификатор ширины поля задается числом между

Модификатор точности задает точность числа с плавающей запятой. Обозначется символом "."и числом после, указывающим количество знаком после запятой. Если модификатор точности используется для целого числа, то задает минимальное количество цифр в числе и в случае недостатка из дополняет число ведущими нулями. Если используется для строки, то задает максимальное количество символов. Если строка окажется длинее, лишние символы будут отброшены.

```
1 pi = 3.14159265359
2 d = 13
3 s = "Hello"
4 day = 8
5 print(string.format("PI - %.4f, number = %.4d, today is
     %02d of July, string with less than 5 symbols - %.4s",
     pi, d, day, s))
```

Результат выполнения: PI - 3.1416, number = 0013, today is 08 of July, string with less than 5 symbols - Hell

7. `string.find` — функция ищет шаблон в строке, то есть ищет подстроку, соответсущую заданному паттерну. Функция возвращает два числа, индекс начала и индекс конца, или `nil`, если совпадений обнаружено не было. Так же есть возможность передать в функцию третий параметр, задающий индекс, с которого будет начат поиск.

```
array = {}
s = 123145167
i = 0
while true do
i = string.find(s, '1', i+1)     -- find next '1'
if i== nil then
        break
end
table.insert(array, i)
end
print(table.concat(array, ","))
```

Результат выполнения: 1,4,7

8. `string.gsub` — функция, которая используется для замены посдтрок, соответвующих шаблону, на заданную строку. Так же как и другие функции со строкам она не изменяет исходную строку, а лишь возвращает новую. Функция принимает три параметра: исходную строку, шаблон и строку для замены, есть возможность передать четвертый параметр, который задает максимальное количество замен. Помимо новой строки функция так же возвращает количество замен, поэтому при помощи этой функции можно быстро подсчитать количество определенных подстрок или символов, например количество пробелов в строке.

```
s="Hello, somebody"
s=string.gsub(s, "llo", "p")
print(s) --> Help, somebody
s=string.gsub("All lla", "l", "x")
print(s) --> Axx xxa
s=string.gsub("All lla", "l", "x", 2)
print(s) --> Axx lla
s, count = string.gsub("Lua is great", " ", " ");
print(count) --> 2
```

9. `string.match` — аналогична `string.find`, то есть так же ищет подстроку, подходящую под шаблон, но вместо индексов конца и начала возвращает найденный текст.

```
print(string.match("foo 123 bar", '%d%d%d')) -- %d matches
    a digit
```

Результат выполения: 123

1.10.1 Шаблоны

Шаблон представлет из себя строку, которая может включать в себя специальные символы, обозначающие опредеенную последовательнось обычных символов.

1. Символьный класс — часть шаблона, обозначающая символ из определенного множества

   (a) . — любой символ

   (b) %a — буквы

   (c) %c — управляющие символы

   (d) %d — целые числа

   (e) %l — буквы в нижнем регистре

   (f) %p — знаки пунктуации

   (g) %s — разделители(пробел, табуляция)

   (h) %u — буквы в верхнем регистре

   (i) %w — буквы и цифры

   (j) %x — шестнадцатеричные цифры

   Можно создать собственный класс, заключив список символов или их диапазон в []. Для указания диапазона между символами ставится знак -, если же этот символ будет стоять первым символом после открывающейся квадратной скобки или последним символом перед закрывающецся квадратной скобкой, то он будет считаться обычным символом. Для создания отрицательного класса после открывающей скобки необходимо поставить символ ^.

```
1 s = "Deadline is 21.08.2016, hurry up"
2 date = "%d%d.%d%d.%d%d%d%d"
3 print(string.sub(s, string.find(s, date))) --> 21.08.2016
4 print(string.match("123a something", "12[34][a-z]")) -->
    123a
5 print(string.match("124z something", "12[34][a-z]")) -->
    124z
6 print(string.match("125y something", "12[34][a-z]")) --> nil
```

2. Квантификаторы — определяют количество возможных появлений впереди стоящего символа

   - `*` — повторение впереди стоящего символа(или класса) ноль или более раз(как можно больше символов захватывается при сравнении с шаблоном — максимальный квантификатор)
   - `+` — повторение впереди стоящего символа(или класса) один или более раз(как можно больше символов захватывается при сравнении с шаблоном — максимальный квантификатор)
   - `-` — повторение впереди стоящего символа(или класса) ноль или более раз(как можно меньше символов захватывается при сравнении с шаблоном — минимальный квантификатор)
   - `?` — повторение впереди стоящего символа(или класса) ноль или один раз

```
1 print(string.match("aaabc", 'a*')) --> aaa
2 print(string.match("aaabc", 'a-')) -->
3 print(string.match("aaabc", 'a+')) --> aaa
4 print(string.match("aaabc", 'a?')) --> a
```

3. Запоминающие скобки позволяют отметить ту часть шаблона, которую необходимо сохранить и затем вернуть, например в функции `string.match`.

```
1 date = "26/8/2016"
2 var1, var2, d, m, y = string.find(date, "(%d+)/(%d+)/(%d+)")
3 print(d, m, y)  --> 26  8  2016
```

4. Специальные символы:

   (a) `^` — начало строки

   (b) `$` — конец строки

   (c) `%` — отменяет специальное значение следующего за ним символа

```
1 print(string.match("Hello 3!, user2!", "[1-9]!")) --> 3!
2 print(string.match("Hello 3!, user2!", "[1-9]!$")) -->2!
3 print(string.match("Hello 3!, user2!", "^[1-9]!")) -->nil
```

1.11. Функции

Функции являются главным механизмом абстракции выражений и утверждений в *Lua*. Они могут как выполнять определенную задачу(процедура или подпрограмма), так и вычислять и возвращать значения. В первом случае функция вызывается как утверждение, во втором как выражение.

```
1 print(8*9, 9/8) --statement
2 a = math.sin(3) + math.cos(10) --expression
```

В обоих случаях пишется имя функции и список аргументов в круглых скобках (). Если список аргументов пустой, то скобки будут пустыми, они нужны для идентификации функции, то есть для определения, что это именно функция. Если функция имеет всего один аргумент, и он является либо строкой, либо конструктором таблиц, то скобки можно упустить.

```
1 print "Hello World" -- print("Hello World")
2 f{x=10, y=20} -- f({x=10, y=20})
```

Также *Lua* предоставляет специальный синтаксис для объектно-ориентированых вызовов — оператор двоеточия. Выражение `o:foo(x)` выполняет абсолютно то же действие, что и выражение `o.foo(o, x)`, то есть является обычным вызовом функции объекта, но в качестве первого аргумента передается объект, функция которого вызывается.

Определение функции всегда содерджит имя этой функции, список параметров и тело, которое включает список выражений:

```
1  function sum_of_array (a)
2        local sum = 0
3        for i,v in ipairs(a) do
4          sum = sum + v
5        end
6        return sum
7     end
```

Параметры работают так же, как и локальные переменные, то есть они видны в пределах функции, но не доступны вне ее. Они инициализируются аргументами, которые подаются при вызове функции. Можно вызвать функцию с количеством аргументов превышающим количество возможных параметров. В случае недостатка аргументов, все оставшиеся параметры будут инициализированы `nil`, в случае избытка, лишние будут проигнорированы.

```
1 function f(a, b)
2 return a or b
3 end
```

CALL PARAMETERS

//* f(3) a=3, b=nil //* f(3, 4) a=3, b=4 //* f(3, 4, 5) a=3, b=4 (5 is ignored)

1.11.1 Множественные значения

Важной особенностью *Lua* является возможность возврата множественных значений из функции, в то время как во многих языках программирования функция может возвращать всего одно значение. Для возврата нескольких значений достатчоно написать их все через запятую после ключевого клова `return`.

Функция, определяющая максимальное значение массива и его индекс:

```
1  function maximum (a)
2        local mi = 1             -- maximum index
3        local m = a[mi]          -- maximum value
4        for i,val in ipairs(a) do
5          if val > m then
6            mi = i
7            m = val
8          end
9        end
10       return m, mi
11     end
```

Функция, которая возвращает несколько значений и вызывается в другом выражении, может вести себя по разному в зависимости от своего местоположения. Например, при множественном присваивании, если вызов функции стоит не последним значением, то тогда будет использовано только первое возвращенное значение. Если же оно стоит на последнем месте в списке, то тогда оно предоставит столько результатов, сколько необходимо.

```
1 function foo2 ()
2  return 'a','b'
3 end
4
5 x,y,z = 10,foo2()    -- x=10, y='a', z='b'
6 x,y,z = foo2()       -- x='a', y='b', z=nil
7 x,y = foo0(), 20, 30  -- x='nil', y=20, 30 is ignored
```

Аналогичное поведение будет наблюдаться, если результат функции передается в качестве параметров в другой вызов или является значением в конструктуре таблиц.

```
1 print(foo2())            -->  a   b
2 print(foo2(), 1)         -->  a   1
3 print(foo2() .. "x")     -->  ax
4
5 a = {foo2()}           -- a = {'a', 'b'}
6 a = {1, foo2(), 4}     -- a[1] = 1, a[2] = 'a', a[3] = 4
```

Можно заставить вызов возвращать только одно значение, если поместить вызов функции в круглые скобки

```
1 function foo (i)
2  return foo2()
3 end
4
5 print(foo(2))      --> a  b
6 print((foo2()))        --> a
```

Стоит отметить, что при помещении возвращаемого значения в варажении `return` в круглые скобки, возвращено будет только одно значение, независимо от того, сколько их на самом деле возвращалось бы. То есть `return (f())` всегда вернет только одно значение - первое.

1.11.2 Переменное число параметров

Некоторые функции в *lua* могут принимать переменное число аргументов, например функция `print`. При создании функции с переменным количеством аргументов достаточно в списке параметров указать троеточие — `...`. Когда функция будет вызвана, переданные параметры, которым не соотвествует конкретное имя аргумента, будут собраны в таблицу, к которой функия может получить доступ по имени `arg`. Помимо самих параметров таблица `arg` имеет дополнительное поле `n`, которое содержит число аргументов в таблице.

Функция, которая сохраняет свои аргументы в глобальную переменную:

```
1 printResult = ""
2
3     function save (...)
4       for i,v in ipairs(arg) do
5         printResult = printResult .. tostring(v) .. "\t"
```

```
6         end
7         printResult = printResult .. "\n"
8       end
```

Иногда функция может иметь некоторое число фиксированных параметров, а затем переменное число параметров.

```
1 function g (a, b, ...)
2 end
```

Примеры:
g(3) a=3, b=nil, arg=n=0
g(3, 4) a=3, b=4, arg=n=0
g(3, 4, 5, 8) a=3, b=4, arg=5, 8; n=2

1.11.3 Именованные аргументы

Механизм параметров в *Lua* является позиционным, то есть первое переданное в функцию значение будет занесено в первый параметр, второе значение во второй и т.д. Но в функциях, которые имеют много различных параметров или одинаковые по типу, но разные по типу параметры, будет удобно передавать аргументы не по позиции, а по имени параметра. В *Lua* для этого достаточно написать имя функции и в фигурных скобках указать параметры с необходимым значением:

Функция, которая переименовывает файл:

```
1 rename{old="temp.lua", new="temp1.lua"}
```

Идея этого метода заключается в том, что мы сохраняем все параметры в одну таблицу, а передаем ее в качестве единственного параметра. По сути мы определеяем новую функцию, которая принимает всего один аргумент и имеет следующее тело:

```
1 function rename (arg)
2  return os.rename(arg.old, arg.new)
3 end
```

1.11.4 Особенности

Функции в *Lua* являются объектами первого класса, то есть они обладают теми же правами, что и примитивные типы, такие как строки или числа. Функцию можно сохранить в переменную, как глобальную, так и локальную, или в таблицу.

Она может быть передана в другую функцию в качестве аргумента, и может быть возвращена из другой функции.

Стоит отметить, что функции являются анонимными, то есть они не имеют имени. Когда мы говорим об имени функции, на самом деле речь идет о переменной, которая содержит функцию.

```
1 p = print -- p refers to print function
2 print(1)
3 print = math.sin -- print refers to sinus function
4 p(print(1))
5 sin = p --sin refers to print function
6 sin(1)
```

Результат работы функции:

1

0.8414709848079

1

Функция создается при помощи специального выражения:

```
1  function foo (x) return 2*x end
```

На самом же деле вышепредставленное выражение является всего лишь “синтаксическим сахаром”, выражением, которое было содано специально для удобства программистов. На самом деле это то же самое, что и запись:

```
1          foo = function (x) return 2*x end
```

Можно сказать, что `function` является типом переменной.

Другой особенностью функций в *Lua* является лексическая область видимости. Данное понятие описывает возможность использования локальных переменных функции внутри других функций, что объявлены внутри нее.

```
1 function sortbygrade (names, grades)
2     table.sort(names, function (n1, n2)
3       return grades[n1] > grades[n2]    -- compare the grades
4     end)
5 end
```

Как уже упоминалось, функции являются объектами, а значит их можно задавать, например, в качестве поля таблицы. Это можно сделать тремя разными способами, которые очевидно вытекают из всего вышесказаного:

```
1. Lib = {}
```

```
Lib.sum = function (x,y) return x + y end
Lib.dif = function (x,y) return x - y end
```

2.
```
Lib = {
  sum = function (x,y) return x + y end,
  dif = function (x,y) return x - y end
}
```

3.
```
Lib = {}
function Lib.sum (x,y)
  return x + y
end
function Lib.dif (x,y)
  return x - y
end
```