

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	9
1.8. Блоки. Глобальные и локальные переменные	9
1.9. Таблицы	10

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://tylernelson.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 языке Lua являются *неизменяемыми*, то есть нельзя обратиться к индексу строки и поменять символ. Объявление строк 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 понадобиться 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 прохождения 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 соответствии 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 производить множественное присваивание, то есть список значений присваивается списку переменных за один шаг. Элементы обоих 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 переменных 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 переменным, которым не представлены значения получают 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 ограничивает логически сгруппированный набор 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`.

```
1 do
2   --some instructions
3 end

1 str = "Hello World"      -- global
2 local x = 10              -- local
```

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

```
1 x = 10
2 i = 1
3
4 while i <= x do
5   local x = i*2          -- local for "while" body
6   print(x)               --> 2, 4, 6, 8, ...
7   i = i + 1
8 end
9
10 if i > 20 then
11   local x                -- local for "then" body
12   x = 20
13   print(x + 2)          --> 22
14 else
15   print(x)               --> 10 (global)
16 end
17
18 print(x)                --> 10 (global)
```

1.9. Таблицы

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 .