



Game Programming with LUA

afewhee@gmail.com





- Functions
- More about Functions
- Data Structure
- LUA Environment
- Module, Object-oriented Programming, LUA Library





5 함수 개요 (Function)





- 함수: 작업의 실행 단위
- 인수 전달: 소괄호를 사용
- 인수가 하나이고 문자열, 또는 테이블 생성자이면 '()'를 생략

```
print(8*9, 9/8)           -- 인수를 전달한 함수 호출
a = math.sin(3) + math.cos(10) -- 수식과 같이 사용되는 함수 호출
print(os.date())           -- 함수의 결과를 인수로 함수 호출
```

()를 생략한 함수 호출 방법

```
print "Hello World"      -- print("Hello World")
dofile 'a.lua'            -- dofile ('a.lua')
print [[a multi-line    -- print([[a multi-line
      message]])          message]])
f{x=10, y=20}             -- f({x=10, y=20})
type{}                    -- type({})
```





```
function '함수 이름' (인수 리스트)  
    do-somthing  
end
```

-- a 배열의 모든 값 더하기

```
function sum (a)  
    local s = 0  
    for i, v in ipairs(a) do  
        s = s + v  
    end  
    return s  
end
```





- C언어는 실 인수, 형식 인수의 개수가 같아야 하지만 LUA 함수는 인수의 숫자가 같이 않아도 함수 함수 호출 가능
- 실 인수 개수가 형식 인수 개수를 초과하면 초과된 인수는 버림

```
function MyFunc(a, b)  
  return a or b or 1  
end
```

호출	인수	결과
<i>MyFunc()</i>	<i>a=nil, b=nil</i>	--> 1
<i>MyFunc(3)</i>	<i>a=3 , b=nil</i>	--> 3
<i>MyFunc(nil, 4)</i>	<i>a=nil, b=4</i>	--> 4
<i>MyFunc(3, 4)</i>	<i>a=3 , b=4</i>	--> 3
<i>MyFunc(3, 4, 5)</i>	<i>a=3 , b=4 (5는 버림)</i>	--> 3





- return 키워드 다음에 반환 값 나열
- return 값1, 값2, ...

ex)

```
s, e = string.find("hello Lua users", "Lua")  
print(s, e) --> 7      9
```

```
function maximum (a)
```

```
    local mi = 1          -- maximum index
```

```
    local m = a[mi]       -- maximum value
```

```
    for i, val in ipairs(a) do
```

```
        if val > m then
```

```
            mi = i
```

```
            m = val
```

```
        end
```

```
    end
```

```
    return m, mi
```

```
end
```

```
print(maximum({8, 10, 23, 12, 5})) --> 23  3
```





-- 다음과 같이 함수가 정의되어 있을 때

```
function foo0 () end          -- returns no results
function foo1 () return 'a' end  -- returns 1 result
function foo2 () return 'a', 'b' end -- returns 2 results
```

```
x, y = foo2()      -- x='a', y='b'
x = foo2()         -- x='a', 'b'는 버림
x, y, z = 10, foo2() -- x=10, y='a', z='b'
```

-- 함수의 결과 값이 없거나 필요한 만큼 반환 받지 못하면 nil로 채움

```
x, y = foo0()      -- x=nil, y=nil
x, y = foo1()      -- x='a', y=nil
x, y, z = foo2()    -- x='a', y='b', z=nil
```

-- 목록에서 마지막 요소가 아닌 함수 호출은 언제나 결과값 하나만 반환

```
x, y = foo2(), 20    -- x='a', y=20
x, y = foo0(), 20, 30 -- x='nil', y=20, 30 버림
```

-- 함수가 다른 함수의 인수로 호출될 때 마지막 인수가 되면 모든 반환 값이 후자의 인수로 전달

```
print(foo0())      -->
print(foo1())      --> a
print(foo2())      --> a  b (함수 호출이 마지막이므로 전부 반환된 값이 인수로 전달)
print(foo2(), 1)   --> a  1 (함수 호출이 마지막이 아니므로 하나만 반환)
print(foo2() .. "x") --> ax
```





-- 여러 개를 받는 경우 테이블 이용

t = {foo0()} -- *t = {}* (an empty table)

t = {foo1()} -- *t = {'a'}*

t = {foo2()} -- *t = {'a', 'b'}*

-- 다음과 같은 경우도 목록에서 마지막 요소가 아니므로 함수 호출은 하나만 반환

t = {foo0(), foo2(), 4} -- *t[1] = nil, t[2] = 'a', t[3] = 4*

-- 함수에서 반환으로 함수 호출하면 전부 반환

function foo (i)

if i == 0 then return foo0()

elseif i == 1 then return foo1()

elseif i == 2 then return foo2()

end

end

print(foo(1)) --> *a*

print(foo(2)) --> *a b*

print(foo(0)) -- (no results)

print(foo(3)) -- (no results)

-- 강제로 결과를 1개만 반환하려면 소괄호로 감싼다. 따라서 return 문에 소괄호를 감싸지 않는다.

print((foo0())) --> *nil*

print((foo1())) --> *a*

print((foo2())) --> *a*

※*return (f(x))*와 *return f(x)*는 차이가 큼





-- *unpack*: 배열을 인수로 받아서 배열의 색인1부터 시작하는 모든 요소를 결과로 반환

```
print(unpack{10,20,30})    --> 10  20  30  
a,b = unpack{10,20,30}    -- a=10, b=20, 30은 버림
```

변경 가능한 함수 *f*를 배열 *a*에 들어 있는 모든 가능한 값들로 호출할 때
f(unpack(a))

```
f = string.find  
a = {"hello world", "ld"}  
b = f(unpack(a))  
print(b)          --> 10
```

-- 재귀 기법으로 *unpack()* 구현

```
function unpack (t, i)  
  i = i or 1  
  if t[i] ~= nil then  
    return t[i], unpack(t, i + 1)  
  end  
end
```





- 가변 인수 지정 "..."

```
function add(...)
  local s = 0
  for i, v in pairs{...} do  --for i, v in pairs(arg) do
    s = s + v
  end
```

```
  return s
end
```

```
print(add(3,4,10,25,12))  -->54
```

-- 가변 인수로 구현한 print

```
function MyPrint(...)
  local Result = ""
  for i,v in ipairs(arg) do
    Result = Result .. tostring(v) .. "\\t"
  end
  Result = Result .. "\\n"
```

```
  print(Result)
end
```

```
MyPrint("Hello", "world", 2002, "World", " cup")
```





-- *select* 함수를 사용한 *var-arg* 나열

function foo(a,b, ...)

local arg = {...}; arg.n = select("#", ...)

<함수 본체>

end

--> 5.1 이후 '#' 이용한 방법

function foo(a,b, ...)

local arg = select("#", ...)

<함수 본체>

end





- Named Arguments: 매개 변수를 인수가 순서가 아닌 인수 이름에 따라 지정하는 방법
- LUA는 이름 있는 인수 문법이 없지만 테이블로 인수를 묶어서 전달하면 Named Arguments 효과를 만들 수 있음

```
-- sudo-code(LUA 에서는 실행 안됨)
```

```
rename(old="temp.lua", new="temp1.lua")
```

```
function rename (arg)
```

```
-- 이름을 바꾸는 함수
```

```
    return os.rename(arg.old, arg.new)
```

```
end
```

```
ren= {old="temp.lua", new="temp1.lua"} -- 테이블로 인수를 묶는다.
```

```
rename(ren)
```

```
-- 윈도우 생성 예)
```

```
function Window (options)
```

```
    -- check mandatory options
```

```
    ...
```

```
    -- everything else is optional
```

```
CreateWindow(options.title,
```

```
    options.x or 0,
```

```
-- default value
```

```
    options.y or 0,
```

```
-- default value
```

```
    options.width, options.height,
```

```
)
```

```
End
```

```
w = Window{ x=0, y=0, width=300, height=200, title = "Lua", background="blue", border = true }
```

```
Window(w)
```





6 함수 고급 활용





- LUA의 함수는 일등급 값(First Class Value) --> 함수를 변수처럼 사용

```
a = {p = print}                -- 함수를 테이블 객체에 저장  
a.p("Hello World") --> Hello World
```

```
print = math.sin               -- print는 math.sin 함수를 참조  
a.p(print(1))                  --> 0.84870
```

```
sin = a.p                      -- sin 함수는 a.p인 print 함수 참조  
sin(10, 20)                    --> 10      20
```

- 테이블과 마찬가지로 함수를 구현하고 이를 변수에 저장 가능

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





- Anonymous Function(익명 함수):
 - ◆ 함수 이름 없이 **function(인수 리스트) 본체 end**로 네포(Nasted)된 함수
 - ◆ 결과 함수로 저장

```
function MyFunc()  
    local a = 10; local b = 20  
  
    return function(a, b) -- 익명 함수  
        return a + b  
    end  
end
```

```
c = MyFunc()          -- 변수에 함수 저장  
print(c, c(10, 20))  --> function, 30
```

```
--네트워크 ip 리스트에 대한 table 객체의 정렬  
network = {  
    {name = "grauna", IP = "210.26.30.34"},  
    {name = "arraial", IP = "210.26.30.23"},  
    {name = "lua", IP = "210.26.23.12"},  
    {name = "derain", IP = "210.26.23.20"},  
}
```

```
table.sort(network, function(a,b)  
    return (a.name > b.name)  
end )
```





-- 미분 함수 정의

```
function derivative(f, delta)
```

```
    delta = delta or 1e-4
```

```
    return function(x)
```

```
        return ( f(x + delta) -f(x))/delta
```

```
    end
```

```
end
```

-- 미분 함수 호출

```
c = derivative(math.sin)  -- 미분 함수의 f를 sin 함수로 지정
```

```
                        -- 변수 c에 저장
```

```
print(c)
```

```
--> function 출력
```

```
print(c(10))
```

```
--> -0.8390443 미분 값 출력
```





-- 미분 함수 정의

```
function derivative(f, delta)
```

```
    delta = delta or 1e-4
```

```
    return function(x)
```

```
        return ( f(x + delta) -f(x))/delta
```

```
    end
```

```
end
```

-- 미분 함수 호출

```
c = derivative(math.sin)  -- 미분 함수의 f를 sin 함수로 지정
```

```
                        -- 변수 c에 저장
```

```
print(c)
```

```
--> function 출력
```

```
print(c(10))
```

```
--> -0.8390443 미분 값 출력
```





- LUA의 정적 범위 지정(Lexical scoping): 함수 안에서 함수를 구현하는 경우 내포된 함수는 내포한 함수의 지역변수를 접근할 수 있음.
- 클로저: 하나의 함수와 그 함수가 정확하게 접근해야 하는 모든 비 지역 변수를 합한 것
→ 자신의 범위밖에 있는 변수를 접근할 수 있음.
- LUA의 함수는 클로저. 함수형 프로그램, callback 함수로 사용하기 용이

```
function newCounter ()  
  local i = 0  
  return function ()  -- 익명 함수  
    i = i + 1          -- 익명 함수가 자신의 범위 밖에 있는 변수 i를 접근  
    return i  
  end  
end
```

```
c1 = newCounter()  
print(c1()) --> 1  
print(c1()) --> 2    -- i가 local 이지만 static처럼 계속 유효
```

```
c2 = newCounter()  
print(c2()) --> 1    -- static과 다른 부분이 해당 클로저에만 유효  
print(c1()) --> 3  
print(c2()) --> 2
```





--gui 클로저

```
function digitButton (digit)  
  return Button{ label = digit,  
    action = function ()  
      add_to_display(digit)  
    end  
  }  
end
```

-- 보안을 위한 io.open 재정의

```
do  
  local oldOpen = io.open  
  io.open = function (filename, mode)  
    if access_OK(filename, mode) then  
      return oldOpen(filename, mode)  
    else  
      return nil, "access denied"  
    end  
  end  
end
```





- 지역 함수: 함수를 지역 변수에 저장 또는 local 예약어 사용

```
local function()
```

```
...
```

```
end
```

- 주의 지역 함수는 재귀 호출에서 문제가 될 수 있음

```
local f = function(n)
```

```
  return n * function(n-1)  -- buggy
```

```
end
```

```
-- fact(n-1)을 컴파일 하는 시점에서 지역함수 f는 정의되지 않았으므로 error
```

```
-- 다음과 같이 변경
```

```
local f
```

```
f = function(n)
```

```
  return n * function(n-1)
```

```
end
```





6.3 자동 꼬리 호출 (Proper Tail Calls)

- Tail call: 함수 호출로 가장한 goto
- 꼬리 호출할 때 추가 스택을 사용하지 않음. --> 꼬리 호출 제거(tail-call elimination)
- 형식: function f(x) return g(x) end

```
function fact (n)
  if n > 0 then
    return fact(n - 1)      -- 꼬리 호출
  end
end
```

*return x[i].foo(x[j] + a*b, i + j) -- return function() 형태이므로 Proper Tail Call*

- 다음은 꼬리 호출이 아님

```
function f (x)
  g(x)
  return
end
```

- 다음 모두도 꼬리 호출이 아님

```
return g(x) + 1    -- must do the addition
return x or g(x)    -- must adjust to 1 result
return (g(x))    -- must adjust to 1 result
```





6.3 자동 꼬리 호출 (Proper Tail Calls)

--꼬리 호출을 사용한 미로 게임

```
function room1 ()  
  local move = io.read()  
  if move == "south" then return room3()  
  elseif move == "east" then return room2()  
  else print("invalid move")  
    return room1()  -- stay in the same room  
  end  
end
```

```
function room2 ()  
  local move = io.read()  
  if move == "south" then return room4()  
  elseif move == "west" then return room1()  
  else print("invalid move")  
    return room2()  
  end  
end
```

```
function room3 ()  
  local move = io.read()  
  if move == "north" then return room1()  
  elseif move == "east" then return room4()  
  else print("invalid move")  
    return room3()  
  end  
end
```

```
function room4 ()  
  print("congratulations!")  
end
```





7 자료구조





- 자료구조: 자료(사실, 값) + 처리 방법
- 테이블: LUA의 유일한 자료구조. 전산학의 모든 자료구조를 처리할 수 있고, 효율적

- 배열: 순차 자료구조. 테이블로 가변적인 배열 표현

```
a = {}    -- 새로운 배열
for i=1, 1000 do
    a[i] = 0
end
```

```
-- 길이 출력
print(table.getn(a))
print(#a)
```

```
a = {}    -- 새로운 배열
for i=1, 1000 do
    a[i] = 0
end
```

- LUA는 인덱스를 1부터 시작

```
-- [-5, 5] 범위로 배열생성
a = {}
for i=-5, 5 do
    a[i] = 0
end
print(table.getn(a))    --> 5
```

- 생성자를 사용 단일 수식으로 배열 초기화
squares = {1, 4, 9, 16, 25, 36, 49, 64, 81}





- 배열의 배열

```
mt = {}          -- 테이블로 행렬 생성
for i=1,N do
  mt[i] = {}     -- 새로운 열(column) 생성
  for j=1,M do
    mt[i][j] = 0 -- 새로운 행(row) 생성 초기화 0
  end
end
end
```

- 1차원 배열로 생성

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

- 희소 행렬(sparse matrix):

- ◆ 행렬의 값이 거의 0으로 채워진 행렬
- ◆ LUA 테이블을 사용하면 nil로 검색하면서 처리 --> 매우 효율적

```
-- 두행렬의 곱. 값이 nil이면 곱셈을 처리하지 않음
function mult(a, rowindex, k)
  local row = a[rowindex]
  for i, v in pairs(row) do
    row[i] = v * k
  end
end
end
```





- 노드가 {자료 + 링크 포인터}를 가지고 한 줄로 연결되어 있는 방식으로 데이터를 저장하는 자료 구조

-- 리스트 정의

list = nil

list = { value = v, next = list }

-- 리스트 순회

local l = list

while l do

print(l.value)

l = l.next

end







- LUA는 전역변수를 환경 테이블에 저장
- 전역 변수를 저장한 _G table 출력
 - ◆ `for n in pairs(_G) do print(n, "Wn") end`
- 환경 변수 설정 함수: `setfenv()`
- 필드를 수정하는 함수: `setfield()`





8.1 동적 명칭을 가진 전역 변수

- 다른 변수에 담긴 문자열을 전역 변수 이름으로 사용하거나 실행 시점에 전역 변수 이름을 만들 필요가 있을 때 패턴을 사용해서 일반화 시킴

```
function getfield (f)
  local v = _G      -- 전역 테이블을 사용하여 시작
  for w in string.gfind(f, "[%w_]+") do
    v = v[w]
    print(v)
  end
  return v
end
```

--[%w_]+ 은 영문자 '_' 가 반복되는 패턴: 전역 변수 테이블에는 '_'로 시작하는 변수도 존재함

```
function setfield (f, v)
  local t = _G      -- 전역 테이블을 사용하여 시작
  for w, d in string.gfind(f, "([%w_]+)(.?)") do
    if d == "." then -- 마지막 필드가 아니라면
      t[w] = t[w] or {} -- 없다면 새로운 필드를 만듦
      t = t[w]          -- 테이블을 얻는다.
    else              -- 마지막 필드라면
      t[w] = v          -- 값을 할당한다.
    end
  end
end
```

-- ([%w_]+)(.?) 은 영문자와 '_'의 조합 패턴에 마지막에 '.' 0번 또는 1번 있을 때를 의미

```
-- 테이블 t에 .x.y 를 생성하고 10d을 배정
setfield("t.x.y", 10)
```

```
print(t.x.y)    --> 10
print(getfield("t.x.y")) --> 10
```





```
-- 존재하지 않은 전역 변수에 대한 접근을 에러로 처리
-- ※ 다음 코드는 lua.exe에서는 중단됨
setmetatable(_G, {
    __index = function (_, n)
        error("attempt to read undeclared variable "..n, 2)
    end,

    __newindex=function (_, n)
        error("attempt to write to undeclared variable "..n, 2)
    end,
})

--a는 선언이 되지 않아 오류 메시지를 출력
a = 1 -->stdin:1: attempt to write to undeclared variable a
```

- 새로운 변수의 선언은 메타메서드를 건너뛰는 rawset() 함수 사용

```
function declare (name, initval)
    rawset(_G, name, initval or false)    -- nil 값 대신 false 값으로 대치
end
```





- LUA의 환경은 전역변수를 사용하므로 전역 변수가 수정되었을 때 전체 환경이 달라짐
- LUA 5 이상은 함수마다 자신만의 환경을 가지도록 구성됨
- 환경 변경 함수 : `setenv()`

```
-- 전역 변수 선언 검사
local declaredNames={}

setmetatable(_G, {

__newindex = function(t, n, v)
    if not declaredNames[n] then
        local w = debug.getinfo(1, "S").what
        if w ~= "main" and w ~= "C" then
            error("attempt to write to undeclared variable " .. n, 2)
        end
        declaredNames[n] = true
    end
    rawset(t, n, v)
end,

})

--test
a= 1
setfenv(1, {})    --현재 환경을 새로운 빈 테이블로 변경
print(a)          -- stdin:5: attempt to call global 'print' (a nil value)
```





- 환경을 변경하면 전역 접근은 새로운 테이블을 사용

```
--_G 이름으로 환경을 변경하고 다시 배정
a = 1                                -- create a global variable
setfenv(1, {_G = _G}) -- 현재 환경 변경
_G.print(a)                        --> nil
_G.print(_G.a)                     --> 1
```

```
-- 상속을 사용한 환경 변경
a = 1
local newgt = {}                    -- 새로운 환경 생성
setmetatable(newgt, {_index = _G})
setfenv(1, newgt)                  -- 생성한 환경 설정
print(a)                           --> 1
```

```
a = 10
print(a)                           --> 10
print(_G.a)                         --> 1
_G.a = 20
print(_G.a)                         --> 20
```





9 모듈, 객체 지향 , LUA 라이브러리





- 모듈: 거의 독립된 기능을 가지면서 교환 가능한 실행단위
- 패키지 : 모듈의 모음
- LUA의 모듈은 require 함수를 통해서 읽고 테이블에 저장하는 단일한 전역 이름으로 이름공간(namespace)처럼 동작
- 모듈 구성: 함수, 상수
- 모듈은 일 등급 값이 아님

- 모듈 호출 방법

```
require "mod"    -- 모듈 호출  
mod.foo()        -- 모듈내의 함수 실행
```

```
local m = require "mod" -- 모듈을 호출하고 변수 m에 저장  
m.foo()
```

```
require "mod"      -- 모듈 호출  
local f = mod.foo() -- 모듈의 함수를 변수에 저장  
f()
```

```
-- io 에 대한 모듈 사용 예  
local m = require "io"  
m.write("Hello world\n")
```





- 테이블을 만들고, 내보낼 모든 함수를 넣고 테이블을 반환함

```
complex = {}                                -- 복소수에 대한 모듈
function complex.new (r, i) return {r=r, i=i} end -- 생성

complex.i = complex.new(0, 1)               -- 허수부에 대한 상수 정의

function complex.add (c1, c2)               -- 덧셈
    return complex.new(c1.r + c2.r, c1.i + c2.i)
end
function complex.sub (c1, c2)               -- 뺄셈
    return complex.new(c1.r - c2.r, c1.i - c2.i)
end
function complex.mul (c1, c2)               -- 곱셈
    return complex.new(c1.r*c2.r - c1.i*c2.i,
                       c1.r*c2.i + c1.i*c2.r)
end
function complex.inv (c)                    -- 역수
    local n = c.r^2 + c.i^2
    return complex.new(c.r/n, -c.i/n)
end
return complex    -- 테이블 반환

-- test
c = complex.add(complex.i, complex.new(10, 20))
print(c.r, c.i)    -->10 21
```





- LUA의 자료구조는 테이블로 구성되기 때문에 원칙적으로 객체 지향 프로그램을 할 수 없고 비슷하게 흉내낼 수 있음
- 객체 자신의 함수 참조: self 또는 this 이용
- **self**: 명령의 수신자(receiver). this와 비슷

v = 10

```
Account = {balance = 0}  
function Account.withdraw (self, v)  
    self.balance = self.balance - v  
end
```

```
a1 = Account; Account= nil      --  
a1.withdraw(a1, 100.00)         -- withdraw 함수 인수에 a1을 전달함
```





- 콜론(':') : 메서드 변수의 접근에 대한 범위 연산자

```
Account = { balance=0,  
            withdraw = function (self, v)  
                          self.balance = self.balance - v  
                          print("Withdraw ", self.balance)  
                        end  
          }
```

```
function Account:deposit (v)  
  self.balance = self.balance + v  
  print("Deposit ", self.balance)  
end
```

```
a1 = Account; Account=nil      --  
a1.deposit(a1, 200.00)  
a1:withdraw(100.00)           -- a1.withdraw(a1, 100.00)과 같음
```





- 수학 함수들:
 - ◆ `sin`, `cos`, `tan`, `asin`, `acos`, `exp`, `log`, `log10`,
 - ◆ `floor`, `ceil`, `amx`, `min`, `random`, `randomseed`

- 모든 삼각 함수는 `radian`으로 동작
- 수학 함수들을 다시 정의 하는 방법

```
-- sin, cos, tan, asin, acos
local sin, cos, tan, asin, acos = math.sin, math.cos, math.tan, math.asin, math.acos
```

```
-- degree, radian 변환
local deg, rad = math.deg, math.rad
```

```
-- sin 함수의 입력 각도를 radian으로 설정
math.sin = function (x) return sin(rad(x)) end
```

```
-- sin 함수의 입력 각도를 degree로 설정
math.asin = function (x) return deg(asin(x)) end
```

- `math.random(n)`
 - ◆ 정수 `[1, n]` 값을 반환. 인수가 없으면 `[0, 1)` 실수 값을 반환
 - ◆ `seed`가 설정되어 있지 않으면 매번 동일한 값 출력
 - ◆ `math.randomseed(seed)`: 난수의 `seed` 값을 지정

```
math.randomseed(os.time())
```





- `insert(테이블, 위치, 원소)`: 지정 위치 원소 추가. 위치가 없으면 맨 끝에 추가

```
t = {10, 20, 30}
```

```
table.insert(t, 1, 15)
```

```
for i, n in pairs(t) do print(n) end
```

- `--getn()` 테이블 원소 수 반환

```
print(table.getn(t))
```

- `sort(t)`: 정렬

```
table.sort(t, f)
```

- `concat()`: 병합





- `byte()`: 구간의 문자를 숫자로 반환
- `char()`: 숫자를 ASCII로 변환
- `len()`: 문자열 길이 반환
- `lower()`: 소문자로 전환
- `upper()`: 대문자로 전환
- `reverse()`: 역순으로 변경
- `sub()`: 구간 `i`, `j` 사이의 문자열 추출. `j=-1` 문자열 마지막 글자
`j=-2` 그 이전 문자
- `gmatch()`: 문자열에서 패턴과 일치하는 모든 부분을 훑어 나감.

-- 문자열 `s` 안의 모든 문자 수거 예제

```
words={}
```

```
for w in string.gmatch(s, "a+") do
```

```
    words[#words + 1] = w
```

```
end
```





9.3.3 문자열 (string) 라이브러리

- `format()`: C의 `printf()` 함수의 포맷 설정과 동일
- `find(대상 문자열, 패턴)`: 문자열 검색. 시작 위치, 끝 위치 두개의 값을 반환
 - ◆ `i, j = string.find("Hi, Hello world", "Hello")`
 - ◆ `print(i, j)` --> 5, 9
- `gsub(대상 문자열, 패턴, 대체할 문자열)`: 문자열 대체
 - ◆ `s = string.gsub("Lua is cute", "cute", "great")`
 - ◆ `print(s)` --> Lua is great
- 패턴: 문자 분류(character class)와 마법문자(magic character)를 이용
 - ◆ 문자 분류: `.`, `%a`, `%c`, `%d`, `%l`, `%p`, `%s`, `%u`, `%w`, `%x`, `%z`
 - ◆ 마법문자: `() . % + - * ? [] ^ $`
 - ◆ 알파벳, 숫자, `_` 패턴: `[A-Za-z0-9_]`
 - ◆ 주민 번호 : `\d{6}-\d{7}`
 - ◆ 16진수: `[A-Fa-f0-9]`





- `open()`: 파일 입출력 핸들 얻음. "r, "w" 모드 필요
- `close()`: 파일 핸들 닫기
- `input()`: 읽기 모드로 파일 오픈
- `output()`: 쓰기 모드로 파일 오픈
- `read()`: 파일을 읽어 들임. *all을 사용하면 전부 읽음

```
local fd = io.open(loadname)  
local slotnum = fd:read(3)  
fd:close()
```

- `write()`: 파일에 쓰기
- `lines()`: line 단위로 읽기
- `close()`: `open()`으로 개방한 파일 닫기
- `seek()`: 파일 포인터 위치 옮기기
 - ◆ "set" 시작 위치, "end" 파일 끝





9.3.4 입출력(io) 라이브러리

-- 텍스트 파일 읽기 연습

fr = io.open ("test.txt", "r") -- 파일 핸들 얻기

while true do

*local line = fr:read("*line")* -- 라인단위로 파일 내용 읽기

if nil == line then -- 읽을 데이터가 없으면 nil 반환

break

end

print(line) -- 라인 내용 출력

end

fr:close() -- 파일 핸들 반환





- `date()`: 날짜
`print(os.date())`
- `time()`: 시간
`print(os.time())`
- `getenv()`: 시스템 환경 변수
`os.getenv("PATH")`
- `execute()`: 프로그램 실행
`os.execute("notepad.exe")`

