

# 11 자료구조



# 11.1 배열

- 자료구조: 자료(사실, 값) + 처리 방법
- 테이블: 루아의 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
```

- 루아는 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}`



# 11.2 행렬과 다차원 배열

- 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
- ◆ 루아 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

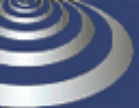
```
local l = list
```

```
while l do
```

```
    print(l.value)
```

```
    l = l.next
```

```
end
```



- 큐: FIFO (First in First Out) 선입 선출 자료구조
  - ◆ 동작: push back, pop front
- Deque(덱): 이중 노드를 사용해서 큐와 스택을 합친 자료구조
  - ◆ 동작: push front, pop front, push rear, pop rear



- 스택: LIFO(Last In First Out) 후입선출 자료구조
  - ◆ 스택 위치를 기억하는 top이 존재
- 동작: push: 원소의 추가, pop : 원소 꺼내오기

-- 스택에대한 자료구조

```
function NewStack ()  
  return {top = 0}  
end
```

Stack={}

```
function Stack.push(lst, value)  -- 스택에 자료 추가(push)  
  local top = lst.top + 1        -- top 위치 1증가  
  lst.top = top  
  lst[top] = value  
end
```

```
function Stack.Dequeue (lst)     -- 스택에서 원소 꺼내오기(pop)  
  local top = lst.top           -- top 위치 1 증가  
  
  if top<0 then  
    print("Stack Under Flow");  
    return  
  end
```

```
  local value = lst[top]  
  lst[top] = nil                -- 초기화  
  top = top -1                  -- top 위치 1 감소  
  return value  
end
```



- Queue: FIFO (First in First Out) 선입 선출 자료구조
- 자료의 위치를 기억하는 front, rear가 존재
- 동작: Enqueue: 원소를 rear에 추가, Dequeue: 원소를 front에서 꺼내오기

```
function NewQueue ()                -- 큐에대한 자료구조
  return {front = 0, rear = 0}
end
```

```
Queue = {}
```

```
function Queue.Enqueue (lst, value) -- 큐의 rear에 insert (Enqueue)
```

```
  local rear = lst.rear + 1        -- rear 위치 1증가
  lst.rear = rear
  lst[rear] = value
end
```

```
function Queue.Dequeue (lst)       -- 큐의 front에서 원소 꺼내오기 (Dequeue)
```

```
  local front = lst.front

  if front >= lst.rear then        -- front 위치가 rear보다 같거나 크면 큐가 비어있는 것
    error("lst is empty")
  end

  local value = lst[front]
  lst[front] = nil                 -- 초기화
  lst.front = front + 1           -- front 위치 1증가
  return value
end
```

- 덱(Deque): 큐의 자료구조에 스택의 동작을 추가한 구조
  - ◆ 자료의 위치를 기억하는 front, rear가 존재
- 동작:
  - ◆ push\_front- 원소를 front앞에 추가
  - ◆ push\_back- 원소를 rear에 추가
  - ◆ pop\_front- front 원소 꺼내오기
  - ◆ pop\_back- rear 원소 꺼내오기

```
function DequeNew ()
    return {front = 0, rear = 0}
end
```

--덱의 자료구조: 큐와 동일

```
Deque = {}
function Deque.push_front (lst, value)
    local front = lst.front - 1
    lst.front = front
    lst[front] = value
    추가
end
```

-- 덱의 front에 insert  
-- front 위치를 1감소 시킴  
-- 1 감소된 front 위치에 원소



```
function Deque.push_rear (lst, value)    -- 덱의 rear에 insert
  local rear = lst.rear + 1           -- rear 위치를 1 증가
  lst.rear = rear
  lst[rear] = value
end
```

```
function Deque.pop_front (lst)         -- 덱의 front에서 원소 꺼내오기
  local front = lst.front
  if front >= lst.rear then
    error("lst is empty")
  end
  local value = lst[front]
  lst[front] = nil                    -- 초기화
  lst.front = front + 1               -- front 위치 1 증가
  return value
End
```

```
function Deque.poprear (lst)          -- 덱의 rear에서 원소 꺼내오기
  local rear = lst.rear
  if lst.front >= rear then error("lst is empty") end
  local value = lst[rear]
  lst[rear] = nil                     -- 초기화
  lst.rear = rear - 1                 -- rear 위치 1 감소
  return value
end
```

## 11.5 집합과 멀티 셋

- 집합(set) 자료구조: 원소 자체가 자료를 구분하는 식별자로 구성된 자료구조, 원소의 중복은 없음.
- 멀티-셋: 원소의 중복이 허락된 셋

*--루아의 예약어를 문자열 키(key)로 사용하는 방법*

```
reserved = {  
  ["while"] = true,    ["end"] = true,  
  ["function"] = true, ["local"] = true,  
}
```

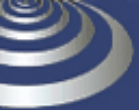
```
for w in allwords() do  
  if reserved[w] then  
    -- 'w' 처리  
  end  
end
```

*-- 좀 더 향상된 방법*

```
function Set (lst)  
  local set = {}  
  for _, l in ipairs(lst) do set[l] = true end  
  return set  
end
```

```
reserved = Set{"while", "end", "function", "local", }
```





- 멀티 셋: 키마다 카운터를 설정, 중복된 원소 만큼 카운터 올림

--멀티 셋 원소 추가

```
function insert(bag, element)
```

```
    bag[element] = (bag[element] or 0) + 1
```

```
end
```

-- 멀티 셋 원소 감소

-- 카운터의 숫자가 1 보다 작으면 nil

```
function remove(bag, element)
```

```
    local count = bag[element]
```

```
    bag[element] = (count and count > 1) and count -1 or nil
```

```
end
```



## 11.6 문자열 버퍼

- 루아의 문자열 변경은 불가.
- 파일에서 문자열을 읽을 때 새로운 문자열을 만들게 되므로 메모리가 증가될 수 있음
  - ◆ `table.concat()`: 주어진 리스트의 모든 문자열을 병합

*-- 파일에 저장된 문자열을 하나의 큰 문자열로 합병*

```
local t = {}  
for line in io.lines() do  
    t[#t + 1] = line .. "\n"  
end  
s = table.concat(t)
```

*-- 제일 작은 문자열 길이부터 큰 문자열 길이 순으로 합병*

```
function addString (stack, s)  
    stack[#stack + 1] = s  
    for i=#stack-1, 1, -1 do  
        if #stack[i+1] < #stack[i] then  
            break  
        end  
        stack[i] = stack[i] .. stack[i+1]  
        stack[i+1] = nil  
    end  
end
```



- 도형으로 표현되는 비 선형 자료구조
- 연결할 객체를 나타내는 정점(Vertex), 이를 연결하는 간선(Edge)의 집합으로 표현

-- 하나의 정점에 간선의 집합으로 표현

*graph = {vertex, edge }*

-- 또는 정점(또는 노드)의 이름과 인접한 노드들의 집합으로 변경

*graph = {name, adj}*

*local function name2node(graph, name)*

*if not graph[name] then*

*-- 노드가 존재하지 않음. 새로운 노드 생성*

*graph[name] = { name = name, adj = adj{ } }*

*end*

*return graph[name]*

*end*

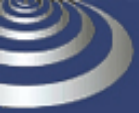


-- 파일로부터 그래프 읽기

```
function read_graph()
  local graph = {}
  for line in io.lines() do
    -- 이름 두 개로 줄을 분할
    local namefrom, nameto = string.match(line, "(%S+)%s+(%S+)" )
    -- 적당한 노드들을 찾는다.
    local from = name2node(graph, namefrom)
    local to   = name2node(graph, nameto)
    -- 'from'에 인접한 집합에 'to' 를 더한다
    from.adj[to] = true
  end
  return graph
end
```

- ※"(%S+)%s+(%S+)" --> 정규표현으로 %S+는 공백이 아닌문자가 1개이상 %s+ 공백문자가 1번이상을 의미





# 12 자료파일 관리와 지속성



# 12.1 자료 파일

- csv 파일 형식: 쉼표(',')로 자료 구분

*Donald E. Knuth, Literate Programming, CSLI, 1992*  
*Jon Bentley, More Programming Pearls, Addison-Wesley, 1990*

- 루아 테이블을 사용하면 자료를 쉽게 구성

## 루아 테이블 1

```
Entry{"Donald E. Knuth",  
      "Literate Programming",  
      "CSLI",  
      1992}
```

```
Entry{"Jon Bentley",  
      "More Programming Pearls",  
      "Addison-Wesley",  
      1990}
```

## 루아 테이블 2 {이름-값}

```
Entry{  
  author = "Donald E. Knuth",  
  title = "Literate Programming",  
  publisher = "CSLI",  
  year = 1992  
}  
  
Entry{  
  author = "Jon Bentley",  
  title = "More Programming Pearls",  
  publisher = "Addison-Wesley",  
  year = 1990  
}
```





## 12.2 직렬화

- 직렬화: 파일 저장 네트워크 전송 등 자료의 처리 순서가 명확히 요구되는 환경에서 자료를
- 바이트 또는 문자 스트림으로 변환하는 작업
- 순환 참조가 없는 단순 테이블 구조 해석
  - ◆ 저장될 스트림을 분석해서 각각 수치, 문자열, 테이블 구조에 대한 코드 저장

```

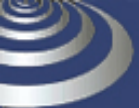
function serialize (o)
  if type(o) = "number" then
    io.write(o)
  elseif type(o) = "string" then      -- 문자열 저장 "%q"는 큰따옴표로 감싸는 것 " -->%"으로 변경됨
    io.write(string.format("%q", o))
  elseif type(o) = "table" then      -- 테이블 스트림은 {} 안에 출력
    io.write("{%#n}")
    for k,v in pairs(o) do

      io.write(" ", k, " = ")        -- first version
      --[[                            --second version
      io.write(" [{"
      serialize(k)
      io.write("] = ")
      --}]

      serialize(v)
      io.write(", %#n")
    end
    io.write("} %#n")
  else
    error("cannot serialize a " .. type(o))
  end
end
end

```





```
-- result of serialize{a=12, b='Lua', key='another "one"'}  
-- first version  
{  
  a = 12,  
  b = "Lua",  
  key = "another #"one#",  
}  
  
-- second version  
{  
  ["a"] = 12,  
  ["b"] = "Lua",  
  ["key"] = "another #"one#",  
}
```



## 12.2 직렬화

- 순환 참조가 있는 테이블 저장
  - ◆ 테이블에 테이블이 있는 구조는 이름이 필요. 이름과 값을 동시에 받아서 처리

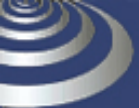
```

function basicSerialize (o)
  if type(o) = "number" then
    return tostring(o)
  else
    return string.format("%q", o)
  end
end

function save (name, value, saved)
  saved = saved or {}
  io.write(name, " = ")
  if type(value) = "number" or type(value) = "string" then
    io.write(basicSerialize(value), "\n")
  elseif type(value) = "table" then
    if saved[value] then
      io.write(saved[value], "\n")
    else
      saved[value] = name
      for k,v in pairs(value) do
        local fieldname = string.format("%s[%s]", name, basicSerialize(k))
        save(fieldname, v, saved)
      end
    end
  else
    error("cannot save a " .. type(value))
  end
end

```



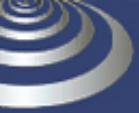


```
--call save  
save('a', a)
```

```
--result  
a = {}  
a[1] = {}  
a[1][1] = 3  
a[1][2] = 4  
a[1][3] = 5
```

```
a[2] = a  
a["y"] = 2  
a["x"] = 1  
a["z"] = a[1]
```





# 13 메타테이블 메타메서드

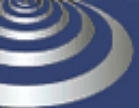


- 메타테이블(metatable):
  - ◆ 루아에서 정의하지 되어 있지 않은 새로운 연산을 지정하는 방법
- 메타테이블 동작:
  - ◆ 두 테이블의 연산에서 메타테이블과 `__` '연산 필드' 필드가 있는지 먼저 확인
  - ◆ 연산 함수가 존재하더라도 메타테이블에 '연산 필드'에 대한 메타메서드가 있으면 이 필드에 정의 된 함수를 먼저 실행
- 테이블에 대해서만 메타 테이블을 적용할 수 있음
  - ◆ 문자열 라이브러리는 문자열 전용 메타테이블만 적용

```
t = {}  
print(getmetatable(t))  --> nil
```

```
-- 메타테이블 적용
```

```
t1 = {}  
setmetatable(t, t1)      -- t 테이블에 t1 테이블을 메타테이블로 적용  
assert(getmetatable(t) == t1)
```



- 산술 연산에 관련된 필드:
  - ◆ '+' : `_add`, '-' : `_sub`, '\*' : `_mul`, '/' : `_div`, '--' : `_unm` ← 부호 반전
  - ◆ '%' : `_mod`, '^' : `_pow`, '..' : `_concat` ← 병합 연산

*-- 집합(set) 자료구조에 대한 메서드 정의*  
*Set = {}*

```
function Set.new (t)
  local set = {}
  for _, l in ipairs(t) do set[l] = true end
  return set
end
```

```
function Set.union (a,b)                                -- 합집합
  local res = Set.new{}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end
```

```
function Set.intersection (a,b)                       -- 교집합
  local res = Set.new{}
  for k in pairs(a) do
    res[k] = b[k]
  end
  return res
end
```



# 13.1 산술 연산 메타메서드

- 산술 연산에 관련된 필드:
  - ◆ '+' : `_add`, '-' : `_sub`, '\*' : `_mul`, '/' : `_div`, '--' : `_unm` ← 부호 반전
  - ◆ '%' : `_mod`, '^' : `_pow`, '..' : `_concat` ← 병합 연산
- 집합의 합집합을 '+' 연산자로, 교집합을 '-' 연산자로 처리하는 방법

```
-- 집합(set) 자료구조에 대한 메서드 정의
Set = {}
```

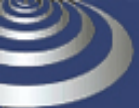
```
function Set.new (t)
  local set = {}
  for _, l in ipairs(t) do set[l] = true end
  return set
end
```

```
function Set.union (a,b)                                -- 합집합
  local res = Set.new{}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end
```

```
function Set.intersection (a,b)                         -- 교집합
  local res = Set.new{}
  for k in pairs(a) do
    res[k] = b[k]
  end
  return res
end
```





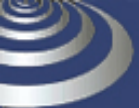


-- 집합 자료구조의 내용을 출력하기 위한 함수

```
function Set.tostring (set)  
  local s = "{"  
  local sep = ""  
  for e in pairs(set) do  
    s = s .. sep .. e  
    sep = ", "  
  end  
  return s .. "}"  
end
```

```
function Set.print (s)  
  print(Set.tostring(s))  
end
```





```
-- '+' 연산자가 호출되면 합집합을 구하도록 메타테이블로 사용할 정규 테이블을 생성
Set.mt = {} -- '+' 연산자를 사용하기 위한 메타테이블
```

```
function Set.new (t) -- 집합 자료구조 생성 함수 수정. 2nd version
  local set = {}
  setmetatable(set, Set.mt) -- 메타테이블 설정
  for _, l in ipairs(t) do
    set[l] = true
  end
  return set
end
```

```
-- Set.new() 로 생성된 모든 집합은 동일한 테이블을 자신의 메타테이블로 지정됨
s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
print(getmetatable(s1)) --> table:
print(getmetatable(s2)) --> table:
```

```
-- 합집합 연산을 위해서 Set.mt에 __add 추가
Set.mt.__add = Set.union
```

```
s3 = s1 + s2 -- '+' 연산자로 합집합 표현
```

```
Set.print(s3) --> {1, 30, 10, 50, 20}
```

```
Set.mt.__mul = Set.intersection -- 교집합 연산을 위해서 Set.mt에 __mul 추가
```

```
Set.print((s1 + s2)*s1) -- +, * 연산자로 합집합, 교집합 연산 --> {30, 10, 50, 20}
```



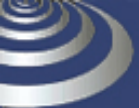
## 13.2 관계 연산 메타메서드

- 관계 연산 필드:
  - ◆ '=': `_eq`, '<': `_lt`, '<=': `_le`
  - ◆ '~', '>', '>='은 다음 관계가 있어 별도의 메타메서드가 없음
  - ◆  $a \sim b \leftrightarrow !(a = b)$ ,
  - ◆  $a > b \leftrightarrow !(a \leq b)$
  - ◆  $a \geq b \leftrightarrow !(a < b)$
- 산술 연산은 다른 타입을 섞어 사용할 수 있으나 관계 연산은 불가
- 항등 연산은 다른 타입을 쓸 수 있으나 다르면 항상 false

*--관계 연산 필드 정의*

```
Set.mt._le = function (a,b)    -- set containment  
  for k in pairs(a) do  
    if not b[k] then return false end  
  end  
  return true  
end
```





```
Set.mt.__lt = function (a, b)  
  return a <= b and not (b <= a)  
end
```

*-- 마지막으로 lt, le를 가지고 항등 비교 연산 정의*

```
Set.mt.__eq = function (a, b)  
  return a <= b and b <= a  
end
```

*-- 메타메서드 정의 테스트*

```
s1 = Set.new{2, 4}  
s2 = Set.new{4, 10, 2}  
print(s1 <= s2)      --> true  
print(s1 < s2)      --> true  
print(s1 >= s1)     --> true  
print(s1 > s1)      --> false  
print(s1 == s2 * s1) --> true
```



## 13.3 라이브러리 전용 메서드

- 연산 함수가 미리 존재하더라도 메타테이블에 '연산 필드'에 대한 메타메서드가 있으면 이 필드에 정의된 함수를 먼저 실행

```
-- print는 언제나 포맷 때문에 tostring() 호출
print({})    --> table:
```

```
-- 만약 연산 필드 _tostring에 대한 메타메서드가
--정의되어 있으면 이 메타메서드를 호출
```

```
Set.mt._tostring = Set.tostring
```

```
s1 = Set.new{10, 4, 5}
-- print() 함수에서 Set의 tostring이 호출됨
print(s1)    --> {5, 10, 4}
```

- setmetatable/getmetatable 함수는 metafield를 사용하지만 이것은 메타 테이블을 보호하기 위한 것

```
-- 메타 테이블을 보호하기 위해 에러나 경고등에 대한 문자 설정
Set.mt._metatable = "메타테이블은 정의할 수 없음"
```

```
s1 = Set.new{}
print(getmetatable(s1))    --> "메타테이블은 정의할 수 없음" 출력
setmetatable(s1, {})      --> 오류 출력
```



# 13.4 테이블 접근 메타메서드

- `__index` 메타메서드: 테이블의 원소 접근 처리
  - ◆ 테이블에 존재하지 않은 인덱스(키) 접근할 때 이를 처리
  - ◆ `index` 메타메서드의 활용은 키가 정해지지 않은 원소를 접근할 때 `nil`을 반환하거나 객체를 새로 생성해서 반환할 수 있는 장점이 있음

```
Window = {} -- 이름 공간 생성
Window.prototype = {x=0, y=0, width=200, height=300,} -- 기본값(Default) 설정
```

```
Window.mt = {} -- 메타테이블 생성
```

```
function Window.new(o) -- 생성자 함수 선언
  print("Call Window.new")
  setmetatable(o, Window.mt)
  return o
end
```

```
-- index 메타메서드 설정
Window.mt.__index = function(table, key)
  print("Call __index")
  return Window.prototype[key]
end
```

```
-- 함수 대신 테이블로 직접 설정
Window.mt.__index = Window.prototype
```

```
-- 테스트
w = Window.new{x=10, y=20}
print(w.width) --> 200 w.width에 접근. w["width"], Window.prototype와 동일
```



# 13.4 테이블 접근 메타메서드

- `__index` 메타메서드: 테이블의 원소 접근 처리
  - ◆ 테이블에 존재하지 않은 인덱스(키) 접근할 때 이를 처리
  - ◆ `index` 메타메서드의 활용은 키가 정해지지 않은 원소를 접근할 때 `nil`을 반환하거나 객체를 새로 생성해서 반환할 수 있는 장점이 있음

```
Window = {} -- 이름 공간 생성
Window.prototype = {x=0, y=0, width=200, height=300,} -- 기본값(Default) 설정
```

```
Window.mt = {} -- 메타테이블 생성
```

```
function Window.new(o) -- 생성자 함수 선언
  print("Call Window.new")
  setmetatable(o, Window.mt)
  return o
end
```

```
-- index 메타메서드 설정
Window.mt.__index = function(table, key)
  print("Call __index")
  return Window.prototype[key]
end
```

```
-- 함수 대신 테이블로 직접 설정
Window.mt.__index = Window.prototype
```

```
-- 테스트
w = Window.new{x=10, y=20}
print(w.width) --> 200 w.width에 접근. w["width"], Window.prototype와 동일
```

- 메타메서드를 함수로 설정하는 것이 테이블보다 비용이 많이 들지만 다중 상속, 캐시 처리, 기타 중간 작업 등을 설정할 수 있어 융통성이 높음



# 13.4 테이블 접근 메타메서드

- `__newindex` 메타메서드

- ◆ 테이블 갱신 처리. 존재하지 않은 인덱스(키)에 값을 설정할 때 처리

```

-- index 메타메서드 설정
Window.mt.__newindex = function (table, key, value)
    print("Call __newindex")
    Window.prototype[key] = value
end

-- 테스트
w = Window.new{x=10, y=20}
w.height = 30          -->w.height에 접근.
print(w.height)       -->30

-- table안에 직접 수정
t = {
    __index = function (temp, k)
        print("*access to element " .. tostring(k))
        return t[k]
    end,

    __newindex = function (temp, k, v)
        print("*update of element " .. tostring(k) ..
            " to " .. tostring(v))
        t[k] = v
    end,
}

function t.new (o)
    print("Call t.new")
    setmetatable(o, t)
    return o
end

w = t.new{}
w[3] = "Hello" --> *update of element 3 to hello
print(w[3])   --> *access to element 3, hello

```





# 13.4 테이블 접근 메타메서드

## ● 테이블 접근 추적하기

- ◆ `_index` 와 `_newindex`를 혼합 사용해서 대행자(proxy)를 만들어 존재하지 않은 키에 대한 처리

```
t = {}           -- 테스트용 원본 테이블
local _t = t    -- 원본 테이블을 비공개 접근 유지하도록 local에 저장
t = {}         -- 대행자(proxy) 생성
```

-- 메타테이블 생성

```
local mt = {
  _index = function (t,k)
    print("*access to element " .. tostring(k))
    return _t[k]           -- 원본 테이블 접근
  end,

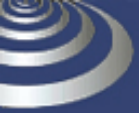
  _newindex = function (t,k,v)
    print("*update of element " .. tostring(k) ..
          " to " .. tostring(v))
    _t[k] = v             -- 원본 테이블 갱신
  end
}
```

```
setmetatable(t, mt)
```

-- test

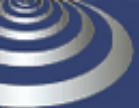
```
t[2] = "hello" --> *update of element 2 to hello
print(t[2])   --> *access to element 2, hello
```





# 14 환경





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



# 14.1 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 만들
```

```
      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
```

## 14.3 비전역 환경

- 루아의 환경은 전역변수를 사용하므로 전역 변수가 수정되었을 때 전체 환경이 달라짐
- 루아 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)
```

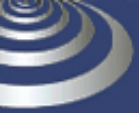


- 환경을 변경하면 전역 접근은 새로운 테이블을 사용

```
--_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
```



# 15-19 모듈, 테이블, 객체지향, 라이브러리





## 15.1 루아 모듈

- 모듈 모듈: 거의 독립된 기능을 가지면서 교환 가능한 실행단위
  - 패키지 :모듈의 모음
  - 루아의 모듈은 require 함수를 통해서 읽고 테이블에 저장하는 단일한 전역 이름으로 이름공간(namespace)처럼 동작
  - 모듈 구성: 함수, 상수
  - 모듈은 일등급 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, i} end -- 생성

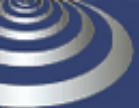
complex.i = complex.new(0, 1) -- 허수부에 대한 상수 정의

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

```





- 루아의 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
```



# 17 약 참조 테이블 (Weak Table)

- 루아는 Garbage Collector가 있어 더 이상 참조가 없는 객체들을 자동으로 수거
- 만약 객체가 집합 (Set)에 있으면 자동으로 수거 안됨
- 약 참조 테이블: 위치에 관계 없이 자동으로 수거될 대상
- 강 참조 테이블: 키와 값 모두 자신이 참조하는 객체의 수거를 방해함으로 strong reference라 함
- `__mode`로 수정: 문자열로 설정
  - ◆ 약 참조 키 설정: `__mode = "k"`
  - ◆ 약 참조 값 설정 : `__mode = "v"`

```
a = {}  
b = {__mode = "k"}  
setmetatable(a, b)      -- a는 약참조 키를 가짐  
key = {}                 -- 첫 번째 키 생성  
a[key] = 1  
key = {}                 -- 두 번째 키 생성  
a[key] = 2  
collectgarbage()        -- 강제로 garbage collector 실행  
for k, v in pairs(a) do print(v) end --> 2
```



- 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
```

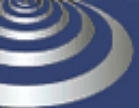
```
-- sin 함수의 입력 각도를 degree로 설정  
math.asin = function (x) return deg(asin(x)) end
```

- `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  

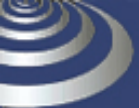
```
print(table.getn(t))
```

- `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={}
```

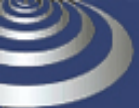
```
for w in string.gmatch(s, "a+") do
```

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

```
end
```

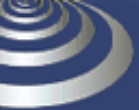






- `format()`: C의 `printf()` 함수의 포맷 설정과 동일
- `find(대상 문자열, 패턴)`: 문자열 검색. 시작 위치, 끝 위치 두개의 값을 반환
  - ◆ `i, j = string.find("Hi, Hello world", "Hello")`
  - ◆ `print(i, j)` --> 5, 9
- `gsub(대상 문자열, 패턴, 대체할 문자열)`: 문자열 대체
  - ◆ `s = string.gsub("Lua is cute", "cute", "great")`
  - ◆ `print(s)` --> Lua is great
- 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]`



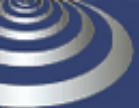


- `open()`: 파일 입출력 핸들 얻음. "r, "w" 모드 필요
- `close()`: 파일 핸들 닫기
- `input()`: 읽기 모드로 파일 오픈
- `output()`: 쓰기 모드로 파일 오픈
- `read()`: 파일을 읽어 들임. \*all을 사용하면 전부 읽음

```
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





-- 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)
```

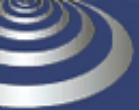
-- 라인 내용 출력

```
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")*

