Scope Tutorial

Variable scopes

Programs are broken up into units of code, like functions and modules. Within these units we may create variables to hold values so that we can process data and make the program perform a given task. For a number of reasons (e.g. potential name clashes, to hide information etc.) we may want to hide variables in one unit from another. We may also want to create temporary variables whilst performing a task which can be removed once we are finished with them. The term "unit" is a little vague. We use the term *scope* to describe the area in which a set of variables live.

Variables that we have access to are said to be *visible*. The scope of a variable is the containing block of code in which it is visible. Scopes are created and destroyed as the program executes and passes in and out of blocks. The variables contained within these blocks are created and destroyed according to the rules described on this page. When we enter a block and a new scope we are entering an *inner scope*. *Outer scopes* are visible from inner scopes but not vice versa.

In Lua blocks of code can be defined using functions and the `do...end` keywords. e.g.,

```
> function foo() local x=1 end
> foo()
> do local y=1 end
```

The above example just defines a function called `foo` which contains a scope. We create the variable `x` in the inner function scope. When we exit the function, the scope ends, the variable `x` is deleted and is no longer visible. The `do...end` block contains similar functionality.

Global scope

Any variable not in a defined block is said to be in the *global scope*. Anything in the global scope is accessible by all inner scopes.

```
> g = "global"
> print(g)
global
> function foo() print(g) end
> foo()
global
```

In the above example `g` is in the global scope, i.e. no enclosing block is defined. The function `foo` is also in the global scope. We enter the `foo` function scope

when `foo()` is called. We can print the value of `g` because we can see the outer scope from the inner `foo` scope.

The local keyword

We use the keyword `local` to describe any variables which we would like to keep local to the scope they are defined in.

Note: All variables declared in Lua revert to the global scope unless they specified local. This may not be the behaviour you are used to. E.g. in C and many other languages variables defined in inner scopes default to that inner scope.

```
> a = 1                     -- global scope
> function foo()            -- start of foo scope
>>  b = 2                    -- no local keyword so global scope
>>  local c = 3              -- local scope
>> end                      -- end of foo scope
> print(a,b,c)              -- before we call foo
1       nil     nil
> foo()
> print(a,b,c)              -- after foo called
1       2       nil
```

Local scope

When we create a block we are creating a scope in which variables can live. e.g.

```
> do local seven = 7 print(seven) end
7
> print(seven)
nil
```

In the above example `do` and `end` enclose a block containing the declaration of the *local variable* `seven`. You can see that we can print its value, 7. When we exit the scope at the `end` keyword we lose visibility of the local variables in that scope. When we print the value of `seven` once outside the scope we get `nil`. This means "variable not found". The keyword `local` is placed before any variable that we want to remain visible only to that scope and its inner scopes.

In the following example `x` starts with the value 1. We create a block using the `do` and `end` keywords. We use the `local` keyword to specify that we want a **new** variable also called `x`, which is only visible in this block or scope.

```
> x = 1
> print(x)
1
```

```
> do
>>  local x
>>  x = 2
>>  print(x)
>> end
2
> print(x)
1
```

You can see that once the `do`...`end` scope has ended the second declaration of `x` disappears and we revert back to the old one.

local in the global scope

The `local` keyword can be used in any scope, not just inner and function scopes. This might seem a little unintuitive but even the global scope in Lua can become a inner scope if it is used as a module.

Because of the implementation of Lua it is more efficient to use `local` variables wherever possible. The technical reason for this is that local variables are referenced via an assigned number, whereas global variables are stored in a table which is accessed with a key (the variable name). Table lookups are very fast in Lua, but still not as fast as local register lookups.

If the following code is compiled we can look at the Lua virtual machine instructions outputted.

```
g = "global"
local l = "local"
print(g,l)
```

The command line tool `luac` (Lua compiler) can be used to compile the code, which gives:

```
main <a.lua:0> (8 instructions, 32 bytes at 00355578)
0 params, 4 stacks, 0 upvalues, 1 local, 4 constants, 0 functions
        1       [2]     LOADK           0 1     ; "global"
        2       [2]     SETGLOBAL       0 0     ; g
        3       [4]     LOADK           0 2     ; "local"
        4       [6]     GETGLOBAL       1 3     ; print
        5       [6]     GETGLOBAL       2 0     ; g
        6       [6]     MOVE            3 0 0
        7       [6]     CALL            1 3 1
        8       [6]     RETURN          0 1 0
```

We won't go into great detail about what is going on here but you can see that the variables `g` and `l` are treated differently, with access to locals being more

immediate. You can find more details in the OptimisationTutorial?.

FindPage · RecentChanges · preferences
edit · history
Last edited November 16, 2003 2:43 pm PDT (diff)