For Tutorial

The `for` statement comes in two different flavours. One can be used to iterate through a numeric progression and the other can be used to iterate over functions called *iterators*. The `for` statement is covered in section 2.4.5 of the Reference Manual.

Numeric progression

The numeric progression version of `for` has the following syntax:

> **`for`** *variable* **`=`** *from_exp* **`,`** *to_exp* **`[`**`,` *step_exp*`]` **`do`** *block* **`end`**

The statement sets the value of the *variable* to *from_exp* before entering the `for` block. The block is only entered if *variable* has not reached the last value, *to_exp*. This includes the first time the loop is iterated over. Each time the block exits *step_exp* is added to *variable*. Specifying the step expression is optional. If it is not specified the value of 1 is used. For example,

```
> for i = 1,3 do print(i) end     -- count from 1 to 3
1
2
3
> for i = 3,1 do print(i) end     -- count from 3 to 1 in steps of 1. zero iterations!
> for i = 3,1,-1 do print(i) end  -- count down from 3 to 1
3
2
1
> for i=1,0,-0.25 do print(i) end -- we're not limited to integers
1
0.75
0.5
0.25
0
```

Note the variable `i` will be local to the scope of the `for` loop. i.e.,

```
> print(i) -- after the above code
nil
```

`for i = e1,e2,e3 do end` is equivalent to the following Lua code:

```
do
  local i, limit, step = tonumber(e1), tonumber(e2), tonumber(e3) or 1
  if not (i and limit and step) then error() end
  while (step>0 and i<=limit) or (step<=0 and i>=limit) do
    -- block code
    i = i + step
  end
end
```

Iterators

The second form of the `for` loop has the syntax:

`for` *var {,* *var}* **`in`** *explist* **`do`** *block* **`end`**

explist is evaluated once before the loop is entered. Its results are an *iterator function* (which sets the *var* values), a *state* (from which the values can be read), and an *initial value* (from which to iterate onwards).

Iterating over tables

If we put a table in place of *explist* Lua will provide the correct *explist* for us. Each element in a table is represented by a *key* and *value* pair. Read the TablesTutorial for more details about usage of tables. To print out all of the elements in a table we can do the following:

```
> table = { 3,7,10,17; banana="yellow", pi=3.14159 }
> for key,value in table do print(key,value) end
1       3
2       7
3       10
4       17
pi      3.14159
banana  yellow
```

pairs(table)

Lua provides a `pairs(table)` function to create the *explist* information for us to iterate over a table. The `pairs()` function will allow iteration over *key-value* pairs.

```
> for key,value in pairs(table) do print(key,value) end
1       3
2       7
3       10
4       17
pi      3.14159
banana  yellow
```

ipairs(table)

The `ipairs()` function will allow iteration over *index-value* pairs. These are *key-value* pairs where the keys are indices into an array:

```
> for index,value in ipairs(table) do print(index,value) end
1       3
2       7
3       10
4       17
```

Notice how only the array part of the table is displayed because only these elements have index keys.

next()

The `next(table [,index])` function helps iterate over a table. When given a table and an index it returns the next *key-value* pair from the table, e.g.,

```
> = next(table) -- index will be nil, the beginning
1       3
> = next(table,"pi")
banana  yellow
```

The `pairs()` function returns an *explist* containing `next()` so we can iterate over tables. We can pass our own expression list to the `for` statement as follows:

```
> for key,value in next,table,nil do print(key,value) end
1       3
2       7
3       10
4       17
pi      3.14159
banana  yellow
```

We pass `next,table,nil` as the expression list to the `for` statement. We are saying here that we want to use the iterator function `next()`, on the table called "table", starting at `nil` (the beginning). The `for` statement keeps executing until the `next()` function returns `nil` (the end of the table).

io.lines()

Lua provides other useful iterators, like `io.lines([filename])` in the `io` library. We can demonstrate this by creating a custom file containing some lines of text.

```
> io.output(io.open("my.txt","w"))
> io.write("This is\nsome sample text\nfor Lua.")
> io.close()
```

We created a file called "my.txt", wrote three lines to it and closed it. Using the `io` line iterator is trivial:

```
> for line in io.lines("my.txt") do print(line) end
This is
some sample text
for Lua.
```

file:lines()

The `io` library provides another way to iterate over lines of a text file.

```
> file = assert(io.open("my.txt", "r"))
> for line in file:lines() do print(line) end
This is
some sample text
for Lua.
> file:close()
```

What are the differences with `io.lines()`?

You have to explicitely open and close the file. One advantage over this is that if the file cannot be opened, you can handle this failure gracefully. Here, the `assert` has the same effect as `io.lines`: the interpreter stops with an error message pointing to the faulty line; but you can test for a `nil` value of `file` and do something else.

Another advantage is that you can start the loop on any line:

```
file = assert(io.open("list.txt", "r"))
local line = file:read()
if string.sub(line, 1, 1) ~= '#' then
  ProcessLine(line) -- File doesn't start with a comment, process the first line
end
```

```
-- We could also loop on the first lines, while they are comment
-- Process the remainder of the file
for line in file:lines() do
  ProcessLine(line)
end
file:close()
```

Custom iterators

We can write our own iterators, similiar to `next()`, to iterate over any data sequence. This is covered in more detail in the IteratorsTutorial.

FindPage · RecentChanges · preferences
edit · history
Last edited April 17, 2004 10:33 am PDT (diff)