Coroutines Tutorial

What are coroutines?

Coroutines allow us to execute several tasks at once. This is done in a controlled manner by passing control to each routine and waiting until the routine says it has finished. We can reenter the routine to continue at a later time and by doing this repeatedly we achieve multi-tasking.

Multi-threading

Each task runs in a *thread* which is separate from the other threads. Having several tasks running at once is often called *multi-threading*. Because there is more than one thread running at once our application is said to be *multi-threaded*.

There are different ways in which multi-threading can be implemented. Some systems allocate a fixed amount of time to each thread and take control away when the time is up, passing control on to the next thread etc. This is called *pre-emptive* multi-threading. In this case each of the threads does not need to worry about how much time it occupies, it is more concerned with its own function.

In other systems, a thread *is* concerned with how long it is taking. The thread knows it must pass control to other threads so that they can function as well. This is is called *cooperative*, or *collaborative* multi-threading. Here, all of the threads are collaborating together to allow the application to operate properly. This is the type of multi-tasking that Lua's coroutines use.

Coroutines in Lua are not operating system threads or processes. Coroutines are blocks of Lua code which are created within Lua, and have their own flow of control like threads. Only one coroutine ever runs at a time, and it runs until it activates another coroutine, or yields (returns to the coroutine that invoked it). Coroutines are a way to express multiple cooperating threads of control in a convenient and natural way, but do not execute in parallel, and thus gain no performance benefit from multiple CPU's. However, since coroutines switch much faster than operating system threads and do not typically require complex and sometimes expensive locking mechanisms, using coroutines is typically faster than the equivalent program using full OS threads.

Yielding

In order for multiple coroutines to share execution they must stop executing (after performing a sensible amount of processing) and pass control to another thread. This act of submission is called *yielding*. Coroutines explicitly call a Lua function `coroutine.yield()`, which is similar to using `return` in functions. What differentiates yielding from function returns is that at a later point we can reenter the thread and carry on where we left off. When you exit a function scope using `return` the scope is destroyed and we cannot reenter it, e.g.,

```
> function foo(x)
>>  if x>3 then return true end  -- we can exit the function before the end if need be
>>  return false                 -- return a value at the end of the function (optional)
>> end
> = foo(1)
false
> = foo(100)                    -- different exit point
true
```

Simple usage

To create a coroutine we must have function which represents it, e.g.,

```
> function foo()
>>   print("foo", 1)
>>   coroutine.yield()
>>   print("foo", 2)
>> end
>
```

We create a coroutine using the `coroutine.create(fn)` function. We pass it an entry point for the thread which is a Lua function. The object returned by Lua is a *thread*:

```
> co = coroutine.create(foo) -- create a coroutine with foo as the entry
> = type(co)                 -- display the type of object "co"
thread
```

We can find out what state the thread is in using the `coroutine.status()` function, e.g.,

```
> = coroutine.status(co)
suspended
```

The state *suspended* means that the thread is alive, and as you would expect, not doing anything. Note that when we created the thread it did not start executing. To start the thread we use the `coroutine.resume()` function. Lua will enter the thread and leave when the thread yields.

```
> = coroutine.resume(co)
foo     1
true
```

The `coroutine.resume()` function returns the error status of the resume call. The output acknowledges that we entered the function `foo` and then exited with

no errors. Now is the interesting bit. With a function we would not be able to carry on where we left off, but with coroutines we can resume again:

```
> = coroutine.resume(co)
foo     2
true
```

We can see we executed the line after the yield in `foo` and again returned without error. However, if we look at the status we can see that we exited the function `foo` and the coroutine terminated.

```
> = coroutine.status(co)
dead
```

If we try to resume again a pair of values is returned: an error flag and an error message:

```
> = coroutine.resume(co)
false   cannot resume dead coroutine
```

Once a coroutine exits or returns like a function it cannot be resumed.

More details

The following is a more complicated example demonstrating some important features of coroutines.

```
> function odd(x)
>>   print('A: odd', x)
>>   coroutine.yield(x)
>>   print('B: odd', x)
>> end
>
> function even(x)
>>   print('C: even', x)
>>   if x==2 then return x end
>>   print('D: even ', x)
>> end
>
> co = coroutine.create(
>>   function (x)
>>     for i=1,x do
>>       if i==3 then coroutine.yield(-1) end
>>       if math.mod(i,2)==0 then even(i) else odd(i) end
>>     end
```

```
>>   end)
>
> count = 1
> while coroutine.status(co) ~= 'dead' do
>>   print('----', count) ; count = count+1
>>   errorfree, value = coroutine.resume(co, 5)
>>   print('E: errorfree, value, status', errorfree, value, coroutine.status(co))
>> end
----    1
A: odd  1
E: errorfree, value, status     true    1       suspended
----    2
B: odd  1
C: even 2
E: errorfree, value, status     true    -1      suspended
----    3
A: odd  3
E: errorfree, value, status     true    3       suspended
----    4
B: odd  3
C: even 4
D: even         4
A: odd  5
E: errorfree, value, status     true    5       suspended
----    5
B: odd  5
E: errorfree, value, status     true    nil     dead
>
```

Basically we have a `for` loop which calls two functions: `odd()` when it encounters an odd number and `even()` on even numbers. The output may be a little difficult to digest so we will study the outer loops, counted by `count`, one at a time. Comments have been added.

```
----    1
A: odd  1       -- yield from odd()
E: errorfree, value, status     true    1       suspended
```

In loop one we call our coroutine using `coroutine.resume(co, 5)`. The first time it is called we enter the `for` loop in the coroutine function. Note that the function `odd()`, which is called by our coroutine function yields. You do not have to yield in the coroutine function. This is an important and useful feature. We return value of 1 with the yield.

```
----    2
B: odd  1       -- resume in odd with the values we left on the yield
C: even 2       -- call even and exit prematurely
```

```
E: errorfree, value, status     true    -1      suspended  -- yield in for loop
```

In loop 2, the main `for` loop yields and suspends the coroutine. The point to note here is that we can yield anywhere. We do not have to keep yielding from one point in our coroutine. We return -1 with the yield.

```
----    3
A: odd  3       -- odd() yields again after resuming in for loop
E: errorfree, value, status     true    3       suspended
```

We resume the coroutine in the `for` loop and when `odd()` is called it yields again.

```
----    4
B: odd  3       -- resume in odd(), variable values retained
C: even 4       -- even called()
D: even 4       -- no return in even() this time
A: odd  5       -- odd() called and a yield
E: errorfree, value, status     true    5       suspended
```

In loop 4, we resume in `odd()` where we left off. Note that the variable values are preserved. The scope of the `odd()` function is preserved during a coroutine suspend. We traverse to the end of `even()`, this time exiting at the end of the function. In either case, when we exit a function without using `coroutine.yield()`, the scope and all its variables are destroyed. Only on a yield can we resume.

```
----    5
B: odd  5       -- odd called again
E: errorfree, value, status     true    nil     dead   -- for loop terminates
>
```

Once again we resume in `odd()`. This time the main `for` loop reaches the limit of 5 we passed into the coroutine. The value of 5 and the `for` loop state were preserved throughout execution of the coroutine. A coroutine preserves its own stack and state while in existance. When we exit our coroutine function it dies and we can no longer use it.
