Scripting Tutorial - Lesson 4

Download supporting files for this tutorial

Texas Instruments TI-Nspire Scripting Support Page

At this point, after three lessons, you should be reasonably comfortable with painting text onto a Lua page, varying the size, style and color of that text, and have some strategies for placing that text where you want it. By storing variables in the names we use for the line numbers, we can easily call and display those variables along with static text.

In this lesson, we go further into Lua by learning how to accept input typed while the page is active. This opens up all sorts of possibilities for creating truly interactive documents.

Lesson 4.1: Inputting text

Lua provides a range of simple and useful ways to capture events of many different types. While the focus here will be on accepting keyboard entry, we can just as easily respond to arrow keys, touchpad and mouse actions. For our purposes here, we will collect the entry of keyboard characters, using **on.charIn(char)**. We will create a variable called **answer** which will be made up of whatever is typed on the keyboard (at least for letters, numbers and regular characters).

```
answer = ""


function on.charIn(char)
      answer = answer..char
      var.store("line3",answer)
-- Refresh the screen after each key is pressed.
      platform.window:invalidate()
end
```

We begin by defining the variable **answer** as an empty string.

When we run the function, **on.charIn**, each character input from the keyboard is added to the variable **answer**. Note the use of the **..** to indicate concatenation, or "glueing together" of strings.

As each character is added to **answer**, it is then **stored** in TI-Nspire as the variable,**line3**, that will be displayed. In this way, we see **line3** built up character by character, as each is typed. Here we introduce the

opposite of **var.recall**, which grabs a TI-Nspire variable. **var.store("var", value)** actually writes a value to a variable in TI-Nspire called, here, "var". If the variable does not exist, this process creates it. Now that is cool.

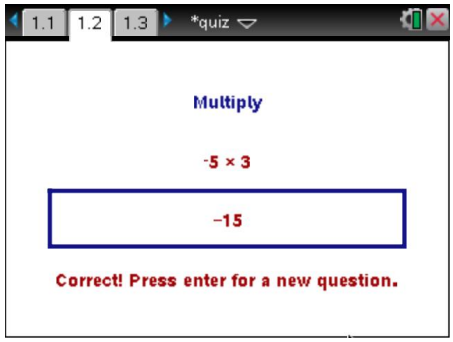
As each character is entered,

the screen is refreshed using **platform.window:invalidate()**. This simply forces the screen to repaint, taking account of any changes that may have just occurred.

```
function on.backspaceKey()

    answer =
    string.usub(answer,0,string.len(answer)-
    1)

    var.store("line3",answer)

    platform.window:invalidate()

end
```

We also want to be able to correct errors, and so enable the backspace key.

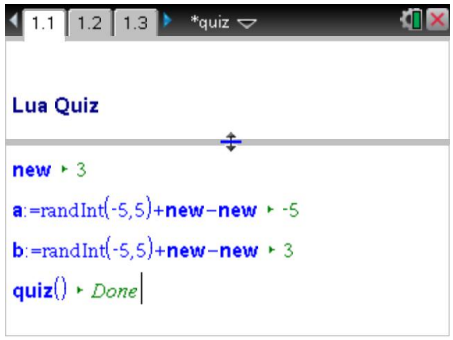
This is a little more involved than adding characters. Instead, we use the string command, **string.usub** which returns the whole of **answer** less one character.



Using these two commands, any text that is typed will be captured as the variable **answer** and displayed in **line3**. Because this is in the standard format that we set up in the last lesson, the input and display of text into **line3** needs no further work. It will now happen automatically. Just add the lines above to your script *before* the **on.paint** function and you may now display questions and other information in lines 1,2 and 4, and any text entered will be shown in **line3**.

Three questions arise from the example shown here: How and where do you enter the values shown on each line? How do you draw that nice blue box around the answer? And how do you get it to recognise when the correct answer has been entered?

Lesson 4.2: Playing Nicely Together



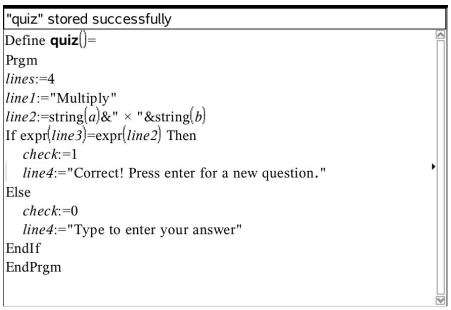
The first of these questions leads to a key point in understanding what is happening here. There is an interplay between the variables in TI-Nspire and those in the Lua page.

Download the supporting files for this lesson, and open and study the document called **script_tut4.tns**. This is a simple algebra quiz document that demonstrates the interplay between variables in TI-Nspire and those in Lua. You should notice that the values for **line1, line2** and so on are easily defined within TI-Nspire. Either using a Notes window (that may then be hidden on a split screen by dragging it down out of sight), or writing a small program and hiding it in a Notes window. Values such as, in our example, the word "Multiply" and the random question are defined within TI-Npire.

For our multiplication example, we generate two random numbers, **a** and **b**. This is best done within a hidden Notes window, even though most everything else will be generated within a little program, here called **quiz()**. If you define random numbers within a more general program, then every

time that program is called, these values will be generated again. In an example like this, every time you make a change to the page, such as entering values, for example, the question will be regenerated. We don't want this to happen – we want to control when a new question is created. To do that, we define a variable called **new** (it doesn't matter what you call it). Then as shown make **new** part of the calculation of the random numbers. A simple way to do this is to add it and then subtract it. In this way, it does not affect the value of the result. However, every time **new** changes then the values of **a** and **b** will also change!

Everything else happens in our little **quiz()** program, as shown.



On page 1.3 we define what we want to appear in lines 1 (for example, line1:=â€Addâ€ or line1:="Multiply") 2 and 4. We do not need to define line3 as this will be our input.

We also define a variable called "check" to be 1 if the answer (**line3**) is correct, or 0 otherwise. Of course, when the answer is correct, the message at the bottom of the screen should be different from otherwise. Notice the condition used here: we take **expr(line2)** (the value of the question) and compare it to the value of the answer typed in **expr(line3)** (the results)

*Both **new** and **check** will actually be picked up by our Lua script and determine what will happen under different conditions, explained below.*

Lesson 4.3: Some Simple Graphics

The inbuilt graphics capability offered by Lua is quite extensive, and will be dealt with in more detail in later lessons. For now, though, it would be neat to be able to highlight our answer. One way to do that would be to draw a box around **line3** as shown on page 1.2 of script_tut4.lua. Study the lines that follow.

```
gc:setColorRGB(20, 20, 138)
gc:setPen("medium", "smooth")

gc:drawPolyLine({0.1*w, h*(linecount-1)/(linecount+1) - 20,
                 0.9*w, h*(linecount-1)/(linecount+1) - 20,
                 0.9*w, h*(linecount-1)/(linecount+1) + 20,
                 0.1*w, h*(linecount-1)/(linecount+1) + 20,
                 0.1*w, h*(linecount-1)/(linecount+1) - 20 })
```

Not surprising to see the graphics context (gc) featured once again, along with **setColorRGB**, and with two new commands: **setPen** (that takes as size inputs such as "thin", "medium" or "thick", and for style, "smooth", "dotted" or "dashed").

drawPolyLine takes a list (table) of x- and y-values to create the polygon required. In this case, 5 ordered pairs define a rectangle, with width and height defined as shown. Look closely at these values and see how they work: vary them a little and see the results.

There is another command in Lua that we will meet called **drawRect** which makes this even easier. It takes four inputs: top left x-coordinate, y-coordinate, length and height of the rectangle. So for this example, we could have defined the box using:

gc:drawRect(0.1*w, h*(linecount-1)/(linecount+1) - 20, 0.8*w, 40)

As Marc pointed out, the height of the rectangle would be much better expressed (like the width) as a multiple of the height of the page (h). This way, if we switch from handheld to computer view, the proportions should remain the same. *Your challenge: What would be a suitable multiple of h for the height of our rectangle?*

Lesson 4.4: Making Enter Work For Us

Wouldn't it be good if, when you press **enter** after getting the correct answer, it creates a new question and wipes the answer box clear, all ready for a new entry?

To achieve this the Lua script needs to know something: It needs to know when the answer is correct. It does this by looking at our variable **check**.

It then needs to generate new values for **a** and **b**. It can do this by sending a new value to the variable **new**. Finally, it will wipe **line3** and the value for **answer** cleared and ready for a new entry.

```
function on.enterKey()
    local ch = (var.recall("check") or 0)
    local newer = (var.recall("new") or 1)
    if ch == 1 then
        var.store("line3","")
        var.store("new",newer+1)
        answer = ""
        var.store("check",0)
    end
    platform.window:invalidate()
end
```

Challenge 1

Suppose we wanted to add something else: What if you were to press **enter** when the answer is *not* correct? At the moment, nothing would happen. But one option could be that this would actually reveal the correct answer. This may or may not be something you want to do with your quiz, but can you see how this might be achieved?

Challenge 2

Suppose we add a slider and call it, say **type**. Minimize it and set it to run from 1 to 4. When **type=1**, the quiz is for *addition*, if 2, *subtraction*, etc. How could this be achieved?

Well that was a big effort! Congratulations on making it to the end of the first section of our Lua tutorials. At this point, there is probably value in reviewing what you have learned so far. Make sure it all makes sense. Play with different commands and try out your own ideas. Then move to lesson 5 in this sequence and try out some useful and cool applications for what we have learned up to this point.

The next sequence of lessons will look at Lua's graphics capabilities, including the use of images. But first, a chance to see a couple of examples of what you might like to do with this stuff!