Lua

STEPHEN KAPLAN

This set of lessons is designed for use in classes, but can also be used for personal use. The lessons will continue to be updated with more information.

Lua Beginning Concepts

- Information
- Getting Started
- Getting user input
- Variables, and how to use them

Information

What is Lua?

Lua is a fast, light-weight, powerful, and easy-to-use scripting language. Lua combines simple syntax with powerful data management. Lua is dynamically typed (Variables and values, expressions, functions, and modules) and has automatic memory management, making it ideal for configuration, scripting, and prototyping.

Why and how is Lua used?

Lua is a robust language. It is constantly being supported and updated. It is fast, meaning it is a good comparison to use when scripting with other languages. So, while learning it is relatively easy, the applications of it are extensive. For example, Lua is currently the leading scripting language for games. It is portable and small, meaning that although it is a powerful, yet simple language, the scripting language is available on most platforms, including mobile computers. Best of all, Lua is free to work with and to learn.

What will we be doing with Lua?

While eventually we would like to make advanced programs from scripting languages, we will start simple. To teach Lua, we will use a ComputerCraft (A mod for Minecraft) Emulator as a learning platform. An emulator lets you have the experience of the ComputerCraft mod without the need to have Minecraft or the mod installed.

- After learning the basics, we will move on to creating slightly more advanced Lua programs (Example: Calculators, Operating Systems, and RedNet (Internet) Applications, such as messengers).

- [If running class] At the start of each lesson, we will review the lesson from the week prior. If wanted, students can share their code and give small talks at the start of the lessons.

- [If running class] Students may present their coding during the time set aside for presenting. They may either share their code, their program, a sample code, or an interesting thing that they have discovered or found online.

SECTION 2

Getting Started

Before We Begin

Please download LOVE from: https://love2d.org

Please download CCEmulator.love from:
https://github.com/Sorroko/cclite/releases/tag/1.0.0

Open The Computer

First, open the ComputerCraft Emulator if you haven't already. Once done, you should see that a world has already been created. If the emulator has not yet been installed on your computer, install the emulator as needed, or ask for help to do so.

Open the emulator and a black interface should appear. This will be our workspace for the next few days. We may later move on to using a more advanced editor and interface for coding Lua. If the black interface appears, you are ready to begin learning.

"Hello World!"

All tutorials normally start with a *Hello World!* program, so we will be doing the same. In the black interface, type *edit* and then a file name, for example: *edit Test*. The text on the interface will change, and you will now be editing a new program or *file*. For starters, we'll learn about printing text with Lua. In the editing interface, type:

```
print("Hello World!")
```

This is considered one line of code, or a *phrase*. Once you have typed this in, the program is complete (Yes, it is quite simple). Press Control, then use the arrow keys to move the brackets to Save button on the bottom left of the interface, and then press enter. Once you have saved the program, press Control again, and now use the arrow keys to access the Exit button. Press enter to exit to the main interface. Type *Test* into the interface (or type the name of the program you created) to run the program. If made and run correctly, the following should appear in the interface:

```
Hello World!
```

Congratulations! You have completed your first program with Lua! As an extra informational bonus, you may want to know that parenthesis and semicolons (at the end of each phrase) are completely optional in Lua, unlike most scripting languages. For example, if you were to leave out parenthesis in Java, you're code may not work at all!

Getting user input

"How old are you?"

Many programs are solely text-oriented, and do not require users to have any real interaction with the script. Those programs can be useful and informational, but not provide a good, enjoyable, or personal user experience. To include the user's response to something, we use the *write* statement. Try something like this:

```
write("How old are you? ")
Age = read()
print("You are " ..Age.. " years old.")
```

Once you have typed this, save and exit the program editing interface. Then type the name of the program. The interface should now display the question:

```
How old are you?
```

It will require you to enter text as a response. When you enter your age, for example, 14, it will respond to your input with the following:

```
You are 14 years old.
```

To hide a user's input by masking it with *asterisks*(*) or *dashes*(-), you can put the symbol you wish to have displayed within the parenthesis of *read()*, like so:

```
write("How old are you? ")
Age = read("*")
print("You are " ..Age.. " years old.")
```

This will display an * for every number, letter, or symbol that a user types in for an input. The actual message that the user types will not be changed, and can be fully displayed later, so the age program would still work and result in the same ending. This masking of text would be useful in creating a login system, so that a user is able to have a secure password.

The first line uses the *write* statement. *Write()* has exactly the same functionality as *print()*, but does not automatically print a new line at the end of the statement. The second line reads a string from the terminal (The interface) and places it as the value of the variable. Variables in Lua are dynamically typed, which means that the variables themselves do not have a type, but the values they hold have a type. In this case, the value stored in name by read is a string. *Values* can have one of six

types in Lua: *nil*, *number*, *string*, *function*, *userdata*, and *table*. The third line prints a message to the person user. The *..* *operator* is the string operator in Lua. Since *Age* contains a string, the result is that the entered age is printed after “You are " on the screen.

Variables, and how to use them

"Tuna = Fish"

Sometimes defining variables at the beginning, or the top, of your document make scripting easier and less confusing. When you define a variable, you can easily use the value of that variable in any part of your code. Variables are case-sensitive, meaning two variables can be the same word, but with different cases, and be considered completely different. You can also change the value of the variable at any point in the code, meaning that you can define it as one value, display that value, then later on, change the value. This would be useful in renaming users and creating new passwords for a login system. To define a variable, type:

```
local variablename = "value"
```

The value of the variable, when it stands for text, is called a *string*. You write *local* before the variable name to tell the program that the variable is within this script or function. While using local variables is not a must-do in ComputerCraft, it is necessary when using Lua for other platforms. Without *local*, the variable would be a *global* variable, and could be altered by other scripts. It is a good habit to use local variables so that you don't make mistakes in the future. For the purpose of this lesson, we will be using the following line of code to work with:

```
local tuna = "Fish"
```

To print the value of a variable, put the variable name inside the parenthesis of a print() or write() phrase.

```
local tuna = "Fish"
print(tuna)
print("Tuna is a "..tuna)
```

In the above code, we defined the variable *tuna*, then printed the variable value normally. We then printed other text along side the variable value. Once again, you see the *..* *operator*. We're saying "Tuna is ", then printing the value of variable tuna.

Variables can have multiple words. They can stand for *numbers*, *symbols*, or *other variables*. If a variable stands for a number, no quotation marks have to be added. For an example of this, see the example:

```
local num1 = 1

local num2 = 2
print(num1 + num2)
print(“The answer is “..num1+num2)
```

SECTION 5

A Few Extra Tips

Terminating, rebooting, and shutting down

To terminate a program on the emulator, press and hold *control+t*. Holding *control+r* will reboot the computer, and holding *control+s* will shut the computer down.

Commenting and Breaking Lines

To makes comments, use "—" before text. This will make the program ignore any text behind the dashes. To "break a line", use "\n". This will cause a new line to be made without the need to make an entirely new phrase.

Commenting: **— This would be a comment**

Breaking Lines: **print("Hi,\nHow are you?")**

"How are you?" would be printed on a line below "Hi", because of the line-break.

SECTION 6

Review

REVIEW 1.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

What would you write before a variable name if you wanted to restrict the use of said variable to only that program?

- **A.** Global
- **B.** Local
- **C.** Restricted
- **D.** NoUse

Check Answer

Lua Beginning Concepts

- Good coding habits
- If, then, else statements
- Input and output codes
- Terminal API

Good Coding Habits

Making Comments

Comments can be boring, seemingly useless, and they add clutter to your code, but they also make your code easy to understand and to edit. Comments make you remember what you've coded, and they serve to tell you, sometimes, why you coded it. Comments can also be used to temporarily disable parts of code. If you're they pre-planning type of person, then comments can also be used to make messages to yourself to make you remember something.

Indenting

When you are working within an if, then, else statement within another if, then, else statement, your code, left unindented, can become confusing and difficult to follow. Indenting the lines within each statement, much like commenting, help give you a sense of context. Indentation can instantly make your code much easier to work with.

Correct Naming

When naming variables and functions, be sure to name everything with a name that fits. Instead of naming something "ThisIsAFunction", name it "Introduction" or "MoveUp". Depending upon the situation you are dealing with, the name of the function may be a label, an action, or a general summarization of what the function or variable does or stands for. When someone, or yourself, looks over your code, they'll know exactly what its purpose is.

Keep the code clean

Keeping your coding clean, organized, and "trash-free" is a task that many people simply forget about. When you're coding, though, you'll realize that keeping useless lines in the code simply makes the work confusing and difficult. Ignoring lines of code that are not being used is so simple, yet getting rid of them improves your code so, so much.

Take breaks, go at your own pace

Do not rush your code. For most people, their best work, be it in school, art, coding, building… anything,

their best work occurs when they are under no stress. Instead of pressuring yourself to finish a code quickly, just stay calm, and get the code done at your own pace. Check dates every now and again just to give you a guideline of where you want to be in your code, but do not feel pressured to stress yourself. If needed, take breaks from your code. Sometimes your best ideas will come from thinking about completely unrelated things. Maybe you'll be playing a game when you think of the next ten lines of code you want to add.

Prototyping, designing, and outlining

One last good coding habit is to prototype, design, and outline your codes before making them. Having an idea of what the code will look like and how it will function is important to know. Knowing how the code will function is good also, but that can come later. When you outline your plan, you're basically just trying to layout the general idea of what your code is. You'll want to have a name, a basic idea of what the code will do, and how the code will be presented. If needed or wanted, you can type or write your ideas. For designing the program, you may want to draw how you imagine it will look. If you're a person who would rather dive right in and code something, go ahead, but some people need the planning prior to making their programs.

If, Then, Else Statements

"If this, then that"

If, then, else statements are incorporated into almost all programs we use. If you click on a button, it does something. If you press enter when typing into a search engine, the search results are shown. If you say "Hi" to a voice recognition program, then the program responds, "Hi". For the purpose of learning, we will use simple math to demonstrate if, then, else statements.

If x is greater than 5, and x is equal to 10, then the statement is true. X, the variable, is equal to ten, which is greater than 5. In Lua, this statement would be produce by the following code:

```
local x = 10
if x > 5 then
    print("Variable X is greater than 5.")
else
    print("Variable X is not greater than 5.")
end
```

As you can see, we declared that local variable x is equal to 10. Being that 10 is a number, no quotation marks are needed. We then asked the program "Is variable x greater than five?". 10 is greater than 5, so the answer is yes. Now that the program knows that x is greater than 5, it will display what we told it to say if x is greater than 5. If variable x was equal to 3, then the "else" part of the statement would come in to action. To close the if, then, else statement, we put "end" at the end of the statement. This is done to indicate that the actions of this statement are done. No code beyond the "end" will be picked up by the statement, meaning that the "else" part can exist without causing trouble. The above code would result in the following being displayed when activated:

Variable X is greater than 5.

SECTION 3

Getting User Input, Then Returning an output

"What is your favorite fruit"

Like we learned last week, we're going to be working with the write() function today. This week, though, we will be learning about how you can take the input that a user gives, and you can give an output based upon what the input is. This can be accomplished by combining if, then, else statements (also known as conditional statements) with reading a user's input with write(). For example, look at this simple code that involves a user inputting their favorite fruit:

```
write("What is your favorite fruit? ")
fruit = read()
if fruit == "Apples" then
    print("Apples are your favorite fruit.")
elseif fruit == "Oranges" then
    print("Oranges are your favorite fruit.")
else
    print(fruit.." are your favorite fruit.")t
end
```

As you can see, this code is a bit more complicated than anything we've done so far, but once described, it is actually very simple. There are multiple additions to this code that we have not told you about yet. In the above code, we're getting the user's input, and calling their input "fruit". The input value is now stored as variable fruit, and can be accessed from calling the variable (..fruit).

The "elseif" part of the statement is also new, but is easily understood. Basically, it is saying that if the other option, fruit equalling "Apples", is not true, but fruit equals "Oranges", then an action will be completed. The else part of the statement remains the same, and is only activated if and when the value of fruit does not match any pre-defined option.

You can embed conditional statements within conditional statements, meaning that you can have tree-like programs that are dynamic and change depending upon what the user does. The multi-conditional statements, as we will call them, can be used in many programs. User-changeable and user-affected content creates an atmosphere in programs that users enjoy and come to expect.

SECTION 4

TERMINAL API

Clearing the terminal

When you clear the terminal, all text is cleared. After all previous information in the terminal has been cleared, any text in your program that will be printed will be displayed alone, and will not be accompanied by any previous information. The clearing the terminal only occurs successively, meaning that it clears only the information that is displayed above it in the code, and does not clear any information displayed after. To clear the terminal, type this line at any point in your code:

```
term.clear()
```

Clearing a line

Like clearing the terminal, you can clear a single line as well. The result is basically the same as clearing the entire terminal, only instead of clearing everything, only one line is being cleared. The line being cleared is the line that the cursor is on. To clear a line, add this text to any part of your code:

```
term.clearLine()
```

Setting the cursor position

When clearing a line, you can either choose to clear the last line that the code has made, or to define an exact line that should be cleared. When you set the cursor position, you can also print new text and display new information on that line. After clearing the terminal, you may want to set the cursor position to the top of the screen. To set the cursor position, add the following line to any point in your program:

```
term.setCursorPos(x,y)
```

The position is defined on an x,y graph. The x would be the line that the cursor is put on, and the y would be how many spaces from the left the cursor would be. You are able to create different designs using the cursor position, and we will be working to create graphical user interfaces soon.

Scrolling

When clearing the terminal is not an appropriate option for your program, you may want to scroll the existing

information in the terminal upwards and create one (or more) lines on the screen. To do this, add the following line to your code where you deem it acceptable:

term.scroll(Number of Lines)

There are more parts to the terminal API, but for now we will just use these ones. Once we begin creating graphic-oriented programs, we will lean how to create colors and blinking text.

A Few Extra Tips

Don’t be afraid of error!

There is no reason to be afraid of errors with lua, because all that will happen is a message will appear telling you exactly what you have done wrong. In a way, it’s actually better to make errors, as you are actually learning from it.

Using Term.Write

While we did not go over this while learning about the terminal API, it is actually quite useful. Term.write() does not create a new line line after the text, however print() does. If you want to separate one line of text into two lines of code, which is sometimes useful (Progress bars for loading screens, for example), simply use term.write() and the job is done for you! Here’s a simple example of the function:

```
term.write(“Loading… “)
term.write(“1”)
    sleep(1)
term.write(“2”)
    sleep(1)
term.write(“3”)
    sleep(1)
term.write(“4”)
    sleep(1)
term.write(“5”)
```

Review

REVIEW 2.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

Do you put quotation marks around a variable's value if it is an integer?

- **A.** Yes
- **B.** No
- **C.** Sometimes

◀ Check Answer ▶

Lua Part I

- Functions
- Shell API
- OS API (os.pullEvent)
- Blocking termination within programs
- Reboot and shutdown programs

Functions

Making Functions

Functions are useful for when you want to use the same part of your code multiple times. Instead of writing the same lines over and over again, you can declare a function, just as you would for a variable. Unlike a variable, however, functions include working lines of code, and can be as long or as short as they want. The code within functions can either be on one line or on multiple lines. Below is an example of a simple function:

```
function draw()
   print(“This is a function!”)
end
```

As you can see, we are using this function to print one-line, or one phrase, or code. For most circumstances, a one-phrase function would be unnecessary. Functions are normally used to include substantial amounts of code that can be accessed throughout an entire program. To call a function, you simple type the second part of the first line above. The word, draw, is the name of the function, and that parenthesis are there for if you wanted the function to work with strings. We’ll get to working with strings for functions in the next lesson. To call the function we’ve made above, you would type this somewhere in your program:

```
draw()
```

This line would be entered anywhere in the code for you program. For example, if you’re working with a conditional statement, you may have someone enter text, and depending upon what they enter, the function would run. This may seem confusing right now, but once you learn it, it’s easy! Let’s take a look at a more advanced program where functions are used:

```
function red()
      term.clear()
      term.setCursorPos(1,1)
      print("Red is a great color!")
end

function ask()
      write("What is your favorite color? ")
      favcolor = read()
      if favcolor == "Red" or favcolor == "red" then
            red()
```

```
        else
            print(favcolor.." is your favorite color.")
        end
end

ask()
```

SECTION 2

Shell API

The shell API is the set of functions in Lua that handles and works with files. Using the shell API, you can accomplish multiple tasks by simply running a code. For now, we'll only be looking at the most basic parts of the shell API. Later, we'll take a look at how you can create, edit, and save programs all from a single code — without having to type the command yourself (For example, instead of typing "edit test", you'd have a program that edits the file, saves it, then runs it.) For the purpose of learning, we'll only be using shell.run() today. While this is, technically, only one function, it has many applications.

Opening files within a program

In some programs, you may want to open another file while running the code. For example, when the user gives a certain answer to a question, you may want to have the program open a different file. Using condition statements, which we learned about in lesson two, you can easily create a program that has many possible options. Each option may lead to a different file, and each file may then lead to a certain answer or piece of information.

```
shell.run("test")
```

The above line of code will open a file, or program, called test. The phrase does nothing more, and depending upon what the file being open does, the program that opens the file can continue. This would be useful in the case that you need to display information to the user, have a quick program run that displays a loading screen (or a different type of visual), and then go back to displaying information to the user.

Performing commands within programs

With shell.run() you can perform any normal command you would give the computer. You can tell the running program to remove a file, copy a file, edit a file, clear the screen, and more. To do this, you would just put the command within the quotation marks.

Below are some examples of the commands you can run within a program using shell.run():

```
shell.run("rm test")
```

This first code will remove a program with the name test.

```
shell.run(“cp test test2”)
```

The above line of code will take a program named test and copy it, and it will then name the copy of the file test2.

Every type of command that you would normally type into the emulator, you can have your code do it as well. There are a great deal of uses for having a code execute commands, such as having a file being renamed depending upon what the user enters. You can use variables within shell.run(), meaning that you can have a user enter text, and then you can use the value of the text they entered as the name of a file being copied. An example of a program that does this can be seen below:

```
write(“What would like the name of the copied file to be? “)
name = read()
shell.run(“cp test “..name)
```

The above program will copy a file named test and have whatever text the user enters be the name of the copy. This is a very simple program, and it could easily be expanded upon to ask a user what file they would like to copy. You would have to ask the user what program they want to copy, have their answer stored as a variable, and then instead of saying (“cp test “..name), you would do (“cp “..firstvariable..” “..secondvariable). In these phrases, the spaces that have been added are needed in order for the program to function correctly. This is because of how you would normally enter the command; with the spaces included. Shell.run() will only run correctly if it is used to run an exact command.

SECTION 3

OS API

The OS API is the set of functions that controls and handles computer events, time (which will be covered later), rebooting and shutting down, IDs, labels, and more. Basically, the OS API handles anythings to do with how the computer operates and what the computer is. The best explanation of this comes with explaining it's individual parts. To start, let's take a look at labels and IDs. These are basic forms of identification for the computers, or emulators, in our case. To view your emulator's ID, type id into the emulator. The ID should be 1 for everyone. This is due to the fact that we're using an emulator instead of the actual mod for a game. For labels, you have to set a label using label set, which will set the name of your computer.

An ID is a constant value, and cannot be changed. A label, on the other hand, can be changed at any point. You can either manually change it, or you can have your code do it.

```
os.setComputerLabel("My Computer")
```

The above line of code is a bit self-explanatory. We're using the OS API, so we start the line by using os. We are then setting the label of the computer, which can be seen in setComputerLabel. Last, we're just naming the computer with text.

```
print(os.getComputerID())
print(os.getComputerLabel())
```

This code will first return the ID of the computer, and then the name of the computer, and will print them both in text-format. This will create an error if no label has been set prior to running this line, but the ID will still be displayed.

Rebooting and shutting down

For some programs, it is necessary to reboot or shutdown the computer. You could, of course, have the user enter "shutdown" and "reboot", but the less dependency that you have on the user, sometimes, the better. Why have the user do something when you can have the code do it for them?

```
os.reboot()
```

The above line of code will cause the computer to reboot, and the running program will end. No code will be executed beyond the point of rebooting.

os.shutdown()

The above line of code will cause the computer to shutdown. Like rebooting, the program will end, and no code will be executed after. Unlike rebooting, shutting down requires the user to hit a button on their keyboard for the emulator to work again.

Both rebooting and shutting down are very simple functions, compared to others. They accomplish only one task each, and neither of them require any other information to function.

Blocking termination within programs

Some programs need a way out for the users, and for those programs, the users can press and hold control+T. Other programs, such as password-entries, should not be able to be terminated at all. Using the OS API, we can block the termination of programs by using a single line of code:

os.pullEvent = os.pullEventRaw

With the above line of code, we're using a built-in function as a variable. os.pullEvent is a built-in function that handles all events that occur. We'll learn more about events in the future. For now, all you need to know is that we're using a function's name as a variable. When you do this, you change what the name or action of the function is. In this case, we're changing both the name and the action for os.pullEvent. When we say that os.pullEvent is equal to os.pullEventRaw, we tell the program that termination is blocked, among other things. This allows for the entering of passwords, usernames, or answers to questions to be required. It blocks all ways of going around the questions, passwords, or usernames. All that is needed to stop users from termination your program is there line of code above.

Like all other variables, changing the value of a function's name will change the function's name (and actions) throughout all programs on the computer, at least until the user reboots the computer. However, unlike regular variables, you cannot use local to keep the change only in one program. This will cause all programs on your computer to become non-terminable, or unable to be terminated, until you restart your computer (or close and reopen the emulator).

Reboot and Shutdown Programs

As you begin to make more advanced programs, you may want to expand the number of files that you work with. Two easy things to do are to create reboot and shutdown programs. These programs, which you must name reboot and shutdown respectively, will replace the default programs for rebooting and shutting down. Within these programs, you can make any code, and whenever a user runs the reboot or shutdown program, the code you write will run. The programs for rebooting and shutting down created by you will not be run if a user uses control+r or control+s to reboot or shutdown the computer.

Reboot Programs

First, we'll start with the rebooting program. All you have to do is create a file called reboot (Type edit reboot to create the file). Now, once you're editing the file, add any code you want. You may want to make an if/else statement that asks the user if they are sure they want to reboot their computer, or you may just want to add some fancy text. To make the computer reboot, you must add os.reboot() to your code. Below is an example of a simple reboot program:

```
write("Are you sure you want to reboot your computer? ")
rb = read()
if rb == "Yes" or rb == "yes" then
    os.reboot()
elseif rb == "No" or rb == "no" then
    print("Ok, canceling reboot.")
else
    print("That's not an answer!")
end
```

As you can see, we're using a simple if/else (conditional) statement, and we're added os.reboot() to when a user enters "Yes" or "yes".

Shutdown Programs

Like rebooting programs, you must create a file called shutdown. Once you're editing the file, you can create any code you want. Use os.shutdown() to shutdown the computer at any point in the code. This will act like rebooting in every way, except for the fact that rebooting makes the computer wake again, while shutting

down requires the user to press a button (key) in order to use the computer once again.

If no shutdown or reboot program exists, the default shut down and reboot operations will be used when a user runs a reboot or shutdown programs. If you create a reboot or shutdown program, it will replace the defaults.

SECTION 5

A Few Extra Tips

Renaming Functions

Like what was discussed when we talked about blocking termination within programs, you can renaming functions. Sometimes, changing the name of a function will also change its actions, depending upon what you're trying to achieve. For now, we'll just talk about renaming the functions.

Suppose you have a function called Start. You can apply that function at any point in your code by using Start(). You may want to rename the function for a variety of reasons, but the reason, for right now, is not important. All you need to know is how you would rename a function. To do this, with the supposed function called Start, type:

```
Start = NewName
```

What we're doing here, as you can see, is renaming the function Start to NewName. Now, whenever the Start function needs to be used, you would type NewName().

Running a program again from within a program

Within some programs, you may need to run the entire program again. For some, a loop would make sense to use. We'll learn about those in a later lesson, though. For now, we'll use the Shell API. To run a program, simply use:

```
shell.run("ProgramName")
```

In this case, ProgramName would be the name of the program currently running. Whenever you have this within your code, you'll run the program again. Using an example above, but with running the program again, you would have this code:

```
write("Are you sure you want to reboot your computer? ")
rb = read()
if rb == "Yes" or rb == "yes" then
    os.reboot()
elseif rb == "No" or rb == "no" then
    print("Ok, canceling reboot.")
```

```
else
    print(“That’s not an answer!”)
    shell.run(“reboot”)
end
```

As you can see, the code will run again each time the user enters an invalid response.

Review

REVIEW 3.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

What is the correct way to reboot the computer?

- **A.** os.r()
- **B.** os.boot()
- **C.** os.restart()
- **D.** os.shutdown()
- ✓ **E.** os.reboot()

◀ Check Answer ▶

Lua Part I

- Colors
- Loops
- Startup Programs
- Simple Login Programs

Colors

Adding Color To Your Programs

Adding colors to your programs offer multiple benefits. Colors can create a more calm, enjoyable experience within your programs. They can add a more “fun” feeling, and the use of the program will instantly become more enjoyable. Adding colors can also be useful in improving the effectiveness of your programs. With colors, you have the ability to highlight key pieces of text or information. You can also make different parts of the program be different colors to indicate what action is being completed at the time. Remember: When you change the background or text colors, it changes for all text or background space that comes after, until you change it again.

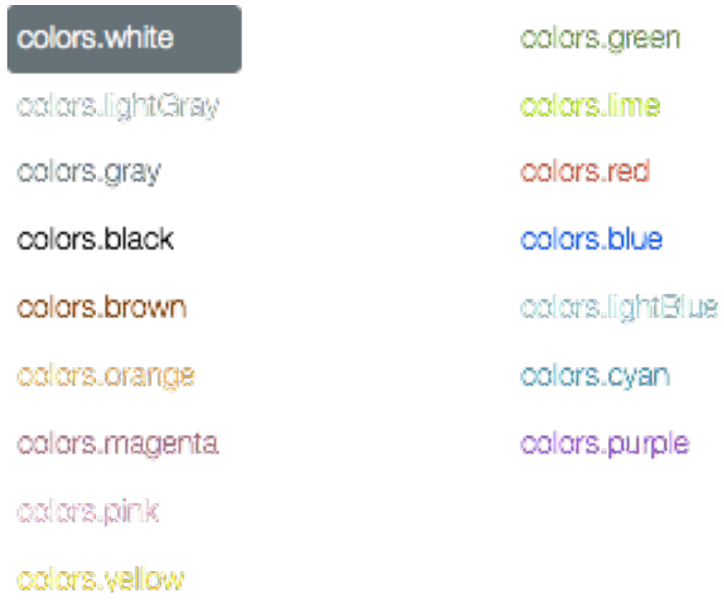
term.setBackgroundColor(colors.color)

The above function is pretty self-explanatory, and its only task is to do exactly what it says. It sets the background color of the text. To make the entire interface one background color, use term.clear() after you set the background color. Likewise, use term.clearLine() after setting the background color to change the background of an entire line.

term.setTextColor(colors.color)

The above function is similar to setting the background color, but it sets the text color instead.

If you want, you can change both the background color and the text color at the same time. You have have white text on a blue background, for example. Simply use one color function(background) and then the other(text). Because of that fact that you can change both at the same time, you’re able to easily make text readable on a background of any color. Remember to always keep the text displayed readable, or else the programs can quickly become unusable. For a list of colors you can use, see below:



Loops

For Loops

For loops are one of the most basic loops available in the Lua coding language. The for loop is a simple section of code that is repeated a set amount of times.

```
local w,h = term.getSize()

for i = 1,10 do
	term.clear()
	term.setCursorPos(w/2,h/2)
	print(i)
	sleep(0.05)
end
```

In the above example, we are setting a variable, called i, that is, at first, equal to one. Where we say ten, it means that the coding within the for loop will be repeated ten times. Each time, the variable i will have the number one added to it. Basically, each time the loop repeats, variable i is equal to i+1, or, like we said before, one is added to i every time the code loops. The code above will clear the screen and print the value of i every half second. In the end, the code will produce the number ten.

For quick reference, at the beginning of the above code, we are settings variables w and h, width and height respectively, to term.getSize(). This means that we are getting both the width and the height of the emulator display and turning those values into readable variables. When we set the cursor position, we are setting the x value, w/2, to half the width of the emulator display. The y value, h/2, sets the location of the cursor (next displayed text) to half the height of the screen. Combined, w/2 and h/2, in the case, will cause the text to be displayed in the exact middle of the display.

Like many things in Lua, certain parts of for loops can be controlled by variables. For example, you can have one variable, named num1, be equal to seven, and a second variable, named num2, be equal to 100. If you say for i = num1,num2, the result will be a loop that repeats itself 93 times, or until variable i is equal to 100. You can use a user's input for this as well, as seen below:

```
write("How many times should I loop? ")
Num = read()
```

```
for i=1,Num do
  print("Looped "..i.." time(s).")
end
```

Repeat Loops

Repeat loops are like for loops in the fact that they have a pre-determined number of times that they will repeat, or loop. Unlike for loops, repeat loops require you to have a line of code that adds one to a variable each time, and then state the point at which you want the loop to stop.

```
local i = 1
repeat
  print(i)
  i = i + 1
until i == 10
```

In the example above, we are setting variable i as equal to one. We are then repeating the code to print the value of i and then add one to i. As seen where we have i = 1+1, we have to type this addition, unlike for loops, which do that addition automatically. Then, to end the repeat, we simply say until i == 10, this means that once we've added one to i enough times for it to be ten, the program will be done. Because it stops when i is equal to ten, the actual number 10 will not be printed. This is because the loop stops exactly when i is equal to ten, not when 10 has been printed. For loops and repeat loops are alike in the fact that they both have a pre-determined number of times that they will execute. Like for loops, you can use user input or variables in repeat loops.

While Loops

While loops are used to run a code multiple times during that time that something occurs. With while loops, you are able to repeat a certain code multiple ways. One of these ways is quite like for loops and repeat loops.

```
local i = 1
while i<10 do
  print(i)
  i = i + 1
end
```

As you can see, the above code is much like a repeat loop. We are setting a value for variable i, then repeating a code until i is equal to ten. For every time the code loops, and i is not equal to ten, we print the value of i, then add one to i. The code is much like a repeat

loop, and the number ten will never actually be printed, as the code only goes until it sees that i is equal to ten, then stops.

This type of while loop, repeat loops, and for loops are useful in many situations. For example, you can use all of these to create a loading screen. You can also just have a program count to a certain number, then stop. If you were to switch a few operations around, and subtract one from i instead of add, then you could make a countdown as well. These loops are good examples of loops that have a definite ending, and will only continue for a set amount of time. There are other types of loops that have no definite ending, and can continue forever with no end.

```
function text()
	print("This program never ends!")
end

while true do --Always loop
	text()
	sleep(0.001)
	text()
end
```

In this code, we have a function named text. In this function, we are simply printing one line of text. We then have a while loop, in which we are running text(), sleeping for 0.001 seconds, then running text() once again. Notice that we say while true do at the beginning of the program. You may be wondering, “while what is true?” Basically, what this is saying is that this loop will never end. It is simply stating that “while this code is running, run this code”. Of course, this should be a bit confusing, but all you need to know at the moment is that the code above will never end. The text() function will run forever, until the user holds down Control+T to terminate the program.

SECTION 3

Startup Programs

When you first open the ComputerCraft Emulator, a script is run that causes the screen to display “CraftOS 1.5” at the top of the screen. Somewhere hidden in the files for the emulator, there is a program called startup. Within that file, there is Lua code that tells the emulator to display that text. If you want, you can create your own startup scripts. Remember the reboot and shutdown scripts we talked about last week? Well, like those, all you have to do to make a startup script is create a file named startup. In the startup program, you can have any code you want. You can display words, colors, have a login program (which we’ll talk about later in this lesson), or anything, really. Here’s an example of some code in a startup program:

```
term.setCursorPos(2,2)
print(“Hello, user!”)
term.setCursorPos(1,4)
```

In the code above, we are simply setting the position of the cursor to 2,2, then printing some text onto the screen. Next, we are setting the cursor position to 1,4, which will create a nice gap between the welcome text and the next command entered.

If you use os.reboot() or shell.run(“startup”), the startup program will be run. This means that you can easily have a program that acts as a loop, without the loops. If you wanted to, you could create a startup file that sleeps for a few seconds, then reboots the computer. When the program reboots, the startup program is run again, creating an infinite loop of rebooting. The applications of this, of course, are quite small. It’s just a useful thing to know, and it’s a bit interesting as well.

Simple Login Programs

Using everything we have learned today and in prior lessons, you can create a good-looking, easy to use login programs that runs whenever the startup program is run (Whenever the emulator is opened). Conditional statements, colors, rebooting, loops, startup programs, and the write() and print() functions are used to create a simple login program below:

```
while true do
term.setBackgroundColor(colors.white)
term.clear()

term.setBackgroundColor(colors.lightBlue)
term.setCursorPos(1,1)
term.clearLine()

term.setBackgroundColor(colors.lightBlue)
term.setCursorPos(1,2)
term.clearLine()

term.setBackgroundColor(colors.lightBlue)
term.setCursorPos(1,3)
term.clearLine()

term.setTextColor(colors.white)
term.setCursorPos(2,2)
print("Login")

term.setBackgroundColor(colors.white)
term.setTextColor(colors.gray)

term.setCursorPos(2,5)
write("Password: ")
pass = read("*")

if pass == "password" then
    term.setBackgroundColor(colors.lightBlue)
    term.setTextColor(colors.white)
    term.setCursorPos(2,2)
    term.clearLine()
    print("Welcome!")
    sleep(1)
    term.setBackgroundColor(colors.black)
    term.clear()
    term.setCursorPos(1,1)
    break
```

```
else
    term.setCursorPos(2,2)
    term.clearLine()
    term.setBackgroundColor(colors.lightBlue)
    term.setTextColor(colors.white)
    term.setCursorPos(2,2)
    term.clearLine()
    print("Incorrect Password!")
    sleep(0.5)
end
end
```

Hopefully, the code above is easy to understand. We use conditional statements, loops, and colors to create a login program that look somewhat good. Feel free to change the colors all you want, and make the code your own. Sometimes, the best way of learning how to code is by looking and studying someone else's.

The code above should all be located within a startup file, so that it is run whenever the emulator is run. Like real computers, you'll have to enter your password each time you open the emulator. If you never tell anybody the password, they can't get in (Supposing you don't share you code with them, either).

A Few Extra Tips

Multiple Comments

Below, there is an example of how you would create a multiple line comment. Multiline comments are a good way of having large amounts of information be stored without it messing up the programs you run. They can also be used to cancel large quantities of code without the need to put two dashes in front of each line. Multiline comments save you from unneeded work and wasted time.

```
—[[
comment goes here.
]]
```

Review

REVIEW 4.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

True or false: The colors.color used in setting colors is actually considered a number.

- ✓ **A.** True
- **B.** False

◀ Check Answer ▶

Lua Review

- Printing, writing, term.write()
- Term API
    - Variables and Functions
    - Rebooting and Shutting Down
    - Conditional Statements and Loops

Printing, Writing, and Term.Write()

Printing

To simply print text to the screen, use the following code:

```
print("Text")
```

Above, we state that we are using the print() function by writing print(). We use quotation marks within the parenthesis to indicate that we are using text, or a string. When the print() function is used, a line is created after the text being printed (The next text will be printed one line below).

Writing

To simply write text to the screen, use the following code:

```
write("Text")
```

With writing, you have the ability to ask the user a question, then wait for an answer (if the function is followed by a read command). If you do not have a read command after the write() function, then write() becomes much like print(), and will do the same action.

Term.write()

Like printing, use term.write() to print text onto the screen. To do this, use the following:

```
term.write("Text")
```

Term.write() has mostly the same properties of print(), but with one difference. Print() creates a new line below the text, whereas term.write() does not. This means that you can have multiple term.write() functions, all creating a different part of a phrase, and have the entire phrase being written on a single line.

Terminal API

Clearing the terminal

When you clear the terminal, all text is cleared. After all previous information in the terminal has been cleared, any text in your program that will be printed will be displayed alone, and will not be accompanied by any previous information. The clearing of the terminal only occurs successively, meaning that it clears only the information that is displayed above it in the code, and does not clear any information displayed after. To clear the terminal, type this line at any point in your code:

```
term.clear()
```

Clearing a line

Like clearing the terminal, you can clear a single line as well. To do this, use the following line of code:

```
term.clearLine()
```

Setting the cursor position

When clearing a line, you can either choose to clear the last line that the code has made, or to define an exact line that should be cleared. When you set the cursor position, you can also print new text and display new information on that line. After clearing the terminal, you may want to set the cursor position to the top of the screen. To set the cursor position, add the following line to any point in your program:

```
term.setCursorPos(x,y)
```

The position is defined on an x,y graph. The x would be the line that the cursor is put on, and the y would be how many spaces from the left the cursor would be. You are able to create different designs using the cursor position, and we will be working to create graphical user interfaces soon.

Colors

When you change the background or text colors, it changes for all text or background space that comes after, until you change it again.

```
term.setBackgroundColor(colors.color)
term.setTextColor(colors.color)
```

The above lines of code a relatively easy to understand; the do exactly what you'd think they'd do. They set the background and text colors. In your packet from last week, all available colors are listed.

Variables and Functions

Variables

To define a variable, type:

```
local variablename = “value”
```

To print the value of a variable, put the variable name inside the parenthesis of a print() or write() phrase.

```
local tuna = “Fish”
print(tuna)
print(“Tuna is a “..tuna)
```

In the above code, we defined the variable tuna, then printed the variable value normally. We then printed other text along side the variable value. Once again, you see the .. operator. We’re saying “Tuna is “, then printing the value of variable tuna. Variables can have multiple words. They can stand for numbers, symbols, or other variables. If a variable stands for a number, no quotation marks have to be added. For an example of this, see below:

```
local num1 = 1
local num2 = 2
print(num1 + num2)
print(“The answer is “..num1+num2)
      print("This program never ends!")
end
```

Functions

Functions are useful for when you want to use the same part of your code multiple times.

```
function draw()
   print(“This is a function!”)
end
```

To call the function we’ve made above, you would type this somewhere in your program:

```
draw()
```

This line would be entered anywhere in the code for you program. For example, if you’re working with a conditional statement, you may have someone enter text, and depending upon what they enter, the function would run.

Rebooting and shutting down

os.reboot()

The above line of code will cause the computer to reboot, and the running program will end. No code will be executed beyond the point of rebooting.

os.shutdown()

The above line of code will cause the computer to shutdown. Like rebooting, the program will end, and no code will be executed after. Unlike rebooting, shutting down requires the user to hit a button on their keyboard for the emulator to work again.

Conditional Statements

"If this, then that"

If x is greater than 5, and x is equal to 10, then the statement is true. X, the variable, is equal to ten, which is greater than 5. In Lua, this statement would be produce by the following code:

```
local x = 10
if x > 5 then
    print("Variable X is greater than 5.")
else
    print("Variable X is not greater than 5.")
end
```

The above code would result in the following being displayed when activated:

Variable X is greater than 5.

"How old are you?"

```
write("What is your favorite fruit? ")
fruit = read()
if fruit == "Apples" then
    print("Apples are your favorite fruit.")
elseif fruit == "Oranges" then
    print("Oranges are your favorite fruit.")
else
    print(fruit.." are your favorite fruit.")t
end
```

The above code will ask the user to enter their favorite fruit, then respond based upon their answer.

Loops

For Loops

For loops are one of the most basic loops available in the Lua coding language. The for loop is a simple section of code that is repeated a set amount of times.

```
local w,h = term.getSize()

for i = 1,10 do
	term.clear()
	term.setCursorPos(w/2,h/2)
	print(i)
	sleep(0.05)
end
```

In the above example, we are setting a variable, called i, that is, at first, equal to one. Where we say ten, it means that the coding within the for loop will be repeated ten times. Each time, the variable i will have the number one added to it. Basically, each time the loop repeats, variable i is equal to i+1, or, like we said before, one is added to i every time the code loops. The code above will clear the screen and print the value of i every half second. In the end, the code will produce the number ten.

Repeat Loops

Repeat loops are like for loops in the fact that they have a pre-determined number of times that they will loop. Unlike for loops, repeat loops require you to have a line of code that adds one to a variable each time, and then state the point at which you want the loop to stop.

```
local i = 1
repeat
  print(i)
  i = i + 1
until i == 10
```

In the example above, we are setting variable i as equal to one. We are then repeating the code to print the value of i and then add one to i. As seen where we have i = 1+1, we have to type this addition, unlike for loops, which do that addition automatically. Then, to end the repeat, we simply say until i == 10, this means that once we've added one to i enough times for it to be ten, the program will be done. Because it stops when i is equal to ten, the actual number 10 will not be printed. This is

because the loop stops exactly when i is equal to ten, not when 10 has been printed. For loops and repeat loops are alike in the fact that they both have a predetermined number of times that they will execute.

While Loops

While loops are used to run a code multiple times during that time that something occurs. With while loops, you are able to repeat a certain code multiple ways. One of these ways is quite like for loops and repeat loops.

```
local i = 1
while i<10 do
  print(i)
  i = i + 1
end
```

As you can see, the above code is much like a repeat loop. We are setting a value for variable i, then repeating a code until i is equal to ten. For every time the code loops, and i is not equal to ten, we print the value of i, then add one to i. The code is much like a repeat loop, and the number ten will never actually be printed, as the code only goes until it sees that i is equal to ten, then stops.

This type of while loop, repeat loops, and for loops are useful in many situations. For example, you can use all of these to create a loading screen. You can also just have a program count to a certain number, then stop. If you were to switch a few operations around, and subtract one from i instead of add, then you could make a countdown as well. These loops are good examples of loops that have a definite ending, and will only continue for a set amount of time. There are other types of loops that have no definite ending, and can continue forever with no end.

```
function text()
	print("This program never ends!")
end

while true do --Always loop
	text()
	sleep(0.001)
	text()
end
```

In the code above, we have a function named text. In this function, we are simply printing one line of text. We then have a while loop, in which we are running text(), sleeping for 0.001 seconds, then running text()

once again. Notice that we say while true do at the beginning of the program. You may be wondering, “while what is true?” Basically, what this is saying is that this loop will never end. It is simply stating that “while this code is running, run this code”. Of course, this should be a bit confusing, but all you need to know at the moment is that the code above will never end. The text() function will run forever, until the user holds down Control+T to terminate the program.

Lua Part II

- Making your own API
- What is an API?
- Why are we making our own APIs?
- Changing the text color
- Centering text
- Centering slow printing text

- Using your API

SECTION 1

Making your own API

What is an API?

Simply put, an API, or application programming interface, is a set of functions can be used to ease the work of programming. When someone uses an API, they are, for the most part, just using code that has already been made in order to create something new. To best understand this, you can think of an API kind of as a man-made tool. The tool itself is already made for you, but you have to use it in order to make something else. In the same way, an API is already made, either by you or someone else, and you use that API to create parts of your program.

Let's look at the terminal API, or term API. Specifically, let's look at term.setCursorPos(x,y). With the terminal API, you simply have to write one line of code to change the cursor position. If the terminal API did not exist, how do you think you would change the cursor position? For now, you don't need to know the answer to that question, but you should be aware of how the terminal API helps. Instead of having to create a large amount of code to do a task, you simply use the terminal API to do something.

Why are we making our own APIs?

Much like functions, APIs can be very useful time-savers, mostly because the APIs in Lua are functions. When we learned about functions a few lessons ago, we talked about how they can make coding much easier. APIs, like normal functions, are able to be used over and over again, and can contain as many or as few lines of code as you want.

In making an API, we achieve the ability to do things with coding on a level of complexity that we've not worked with before. We also learn many valuable aspects to the next section of our coding course: intermediate Lua. A few things that we cover while creating and working with APIs:

- Functions (With arguments)
- Locations (Cursor Position, number values, ect)
- Math (In terms of Lua code)
- Colors and text (And using them efficiently and effectively)

- Correct naming of variables and functions

Also in the lesson, we will begin to move towards our goal to create a program (Most likely an operating system) with a mouse-based GUI (Graphical user interface). We will not be working with the direct components to this goal yet, but we will be starting to work with the APIs and functions that will help us later in our efforts to make great programs.

Changing the text color

To start, we'll just focus on a simple API that changes the color of the text. For now, we'll just use red as the color of choice.

```
function red(str)
    term.setTextColor(colors.red)
    print(str)
end

term.clear()
term.setCursorPos(1,1)
red("Hello!")
```

In the above code, we have a function that is appropriately named red. Within that function, we are setting the text colors to red, the printing a variable called str. You may be wondering what str means, and you should. Str is just Lua's way of saying string. In this case, str is whatever you put between the two parenthesis when you call the function red().

Now that we've made this functions, which, in this case, will be our small API, we can go on with our code. First we clear all text and information on the screen, then we're setting the cursor position to (1,1), or the the top-most and left-most part of the screen. After the cursor position is set, we call the function red(), only unlike normal functions, the parenthesis has quotation marks within them, and the quotation marks contain text. This text between the quotation marks is what was being talked about earlier — it is what the variable str is.

Now that we know what str means, we can understand what this function (Or API, in this case) does.The function is reading what text you put between the two quotation marks. Knowing this, you can also see that since the function changes the text color to red, then prints str, the "Hello!" text will be printed in the color red. Remember: Since setting the color of text changes the color of all text that comes after, you must remember to change the color back once you are no longer in need of

a certain color. This is also true for changing the background color.

For the most efficiency, it is better to only use and create functions or APIs when you plan on using them more than once. If you create an entire function to change the color of one piece of text, then you've spent valuable time on something that could've just been done with one line of code. Likewise, if you do not create a function to change the color of text, and you instead use term.setTextColor(colors.color) each time, you've wasted just as much — if not more — time.

Centering text

Next, let's take a look at another relatively simple API (or function) that incorporates math, locations, and text.

```
local w,h = term.getSize()

function cPrint(str,ypos)
    term.setCursorPos(w/2 - #str/2 + 1,ypos)
    print(str)
end

term.clear()
cPrint("Hello",2)
term.setCursorPos(1,h)
```

Above, we have another few lines of code. The first part of this code should be very familiar. We have two variables, w and h, which stand for width and height respectively. Both of these variables are equal to term.getSize(), meaning that w is equal to the width of the screen, and h is equal to the height. Next, we have a function called cPrint. As you can see when you look in the parenthesis, the function looks for two things within itself: a string and a y coordinate (ypos). We've already talked about str, so that shouldn't be too hard to understand, but we'll talk about what ypos is in a second.

Within the function cPrint, we're setting the cursor position to a strange line of code. To make learning easy, let's just look at each individual part of it. The first part of the setting cursor position, the x coordinate, is w/2 - #str/2 +1. With this, we're saying to take the width of the screen and divide it by two, then we subtract from that the numerical value of str, or the length of the string that we're printing, divided by two. Last, we add one to the final number. If done correctly, this part of the position will result in the text being printed in the center of what-

ever line you decide to put it on, which is defined by ypos.

Ypos is simply a variable that stands for the y coordinate in term.setCursorPos(x,y). In this case, we have a function that looks for both a string and the y coordinate when it's used. This means that when we use cPrint("Hello",2), we're having the text "Hello" be printed in the center of the screen on line two.

Now we've declared the function, so let's take a look at the rest of the code. We clear the screen, print the text "Hello" in the center of line two, and then we set the cursor position to (1,h), or the bottom-most, left-most part of the screen. The reason for doing this last part is just to create a slightly better looking program, but it's not necessary.

Centering slow printing text

Along with simply printing a line of text in the center on a line, you can use other text-making functions as well. You can use printing, writing, or textutills.slowPrint(), which adds a slow-printing effect to your text.

```
local w,h = term.getSize()
function cSlow(str,ypos)
    term.setCursorPos(w/2 - #str/2 + 1,ypos)
    textutils.slowPrint(str)
end

term.clear()
cSlow("Hello",2)
term.setCursorPos(1,h)
```

For the most part, the above code is the same as the last one we talked about. This code uses textutils.slowPrint() rather than the normal printing that is commonly used. Above all, this slow-printing function is used to create a slight wait time or a visual effect.

What can you think of?

Much of coding is about making something that you can think of become real. With what we've learned today and in past lessons, feel free to make something amazing. What can you do with what we've learned so far? What kinds of cool programs are you able to make? Could you make a highly useful API?

As we learn more and more about coding, anyone is welcome to share their creations. Already, there have been some great programs being made, and I believe

that there will be many more awesome programs that all of you create. To get you thinking about something you want to make, here are a few ideas along with the coding subjects that apply to them:

- A calculator program (Math, conditional statements, variables, functions)

- Login program (Conditional statements, variables, functions, loops, os API)

- “Artificial intelligence”, input and output program (Conditional statements, variables, functions, math, loops)

- Games (Conditional statements, variables, functions, math, colors, loops)

- Family Feud (Conditional statements, variables, functions, colors, loops)

Using Your API

What good is creating an API unless you actually use it? If you create an API that you’re not using in any of your codes, then you should do away with that API. It’s good to have the APIs that you need, but it’s even better to have only the APIs that you are actually using. In one way, you’d expect more API and functions to be good — they are there, just in case you ever need them — on the other hand, having APIs and functions that you don’t need is a bad thing — the wasted lines of code are making your code more difficult to understand.

Apart from the simple “cleaning your code” that should be done often, you’re APIs that we’ve created today should work perfectly. If they don’t work, ask for help. If the APIs we’ve created today are working for you, then feel free to start making something with it. Start applying the API you’ve made to more of your programs.

If you’ve put local in front of the functions, or, in other words, you’ve made your API functions local functions, then you’ll only be able to use the APIs within that one program. If you’ve not made the functions local, then you’ll be able to use the APIs in all programs (as long as the original program with the functions has been run at least once). It’s best, I find, to put these types of APIs within a startup program, so that the functions are always there for you to use in any and all programs. In the original program containing the functions has not been run yet, you’re program will give you an error when you try to run the function.

Although inefficient, you can also make a copy of the API at the top of each program (You'd have to put local in front of the functions for this to be at all beneficial). This would allow you to have custom differences in the APIs for each program. If you'd rather have easy, no-more-work-needed APIs that you create, you can just put the functions at the top of a startup file (Do not put local in front of the functions, or else this will not work).

An Extra Tip

Recreating already existing APIS

If you want, you can recreate APIs that already exist within Lua. You'd do this for multiple reasons, including the want to have a customized API set that is tailored to your needs. You might just want to make the names of a few things shorter. For example, you might want to shorten term.setCursorPos(x,y) to just pos(x,y). You might want to change textutils.slowPrint() to slow(). There are many difference things that you can do using functions, and remaking, making, and using APIs is one of the best things that you can do with them. Here's an example of a pos() function, which you can probably guess what it does:

```
function pos(xpos,ypos)
    term.setCursorPos(xpos,ypos)
end
term.clear()
pos(15,6)
print("Hello")
```

Review

REVIEW 6.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

What is an API?

- **A.** A unique variable.
- **B.** A set of functions to accomplish a basic task.
- ✓ **C.** A set of functions that can be used to ease the work of programming

◀ Check Answer ▶

Lua Part II

- Using your API in your programs
- Tables
- Text-based GUIs

SECTION 1

Using your API in your programs

When we make APIs, our aim is to use them appropriately, efficiently, and effectively. Like we said last week, if you have a function, variable, or API, and you're not really using it in your program(s), then that API is relatively useless for you. Keep in mind that after this lesson, tables should be added to the list of things that are, basically, useless unless used more than once. The core idea here is that if you must keep your code clean, or else it may become difficult to understand, which may then lead to errors being difficult to find and correct.

Tables

Simple tables

Tables are probably the most difficult to understand part of coding that we have dealt with thus far. Not only are tables confusing, but they can be (and are) needed for all large projects. You can use tables for a number of different things, ranging from just keeping a list of values to creating configuration files for large projects. For now, we'll try to keep it as easy to understand as possible.

```
local names = {"Andrew", "Alfred", "Alex", "Amber", "Alice"}
print(names[1])
```

In the above code, it would at first appear that we are creating a localized variable, but we are not. It is true that local will have the same effect upon tables as it would variables or functions, but apart from that, tables are quite different from anything we've worked with so far. As you can see, we have the name of the table, which is appropriately titled names. Just like functions or variables, it is important to name your tables with names that relate to what the table is being used for. In this case, we have a table, called names, that holds a list of names.

Like variables, we give tables a name followed by an equals sign. After the equals sign, things begin to get different. With many coding languages, tables and datasets are defined by french brackets { and }. In Lua, brackets have (roughly) the same use. Within the brackets, we have multiple strings (the text inside quotation marks) that are separated by commas. Like variables, each of the strings can be as long or as short as you'd like. As you can see, we've made each string a single word, with each word being a different name. In many languages, the first string in a table would be references by the number 0 (In this case, Andrew would be 0). In Lua, the first string in a table is referenced by the number 1 (Andrew would be 1). After the first string, the numbers increase by one each time to reference the next string in the set. Now that we know what numbers we use to represent each string in a table, we can take a look at how exactly you'd print a value in the table.

```
print(names[1])
```

In this line of code, we are simply printing some text. Within the parenthesis, we do not have quotation

marks, just as if we were printing only a variable. Instead of printing a variable, though, we are printing a value from the table that we made earlier. To do this we use names[1], where names is the name of the table that we created, and 1 is the value (string) that we are referencing, or trying to obtain. We use normal brackets around 1 to indicate that it is a value in the table.

You can change the number within the normal brackets to reference any of the values within the table. In this case, you can use 1,2,3,4, and 5. 1 would return the text Andrew, and 5 would return the text Alice. As you can most likely see, creating a table to contain a set of information can save time and space within your code.

Simple tables (Continued)

```
local names = {“Andrew ", “Alfred ", “Alex ", “Amber ", “Alice "}
print(names[1]..names[2])
```

Above, we have another (nearly identical) table. It operates in the exact same way as above. This time, though, we have a space after each word in every string. When the string is referenced, it will have a space directly after it, meaning that we can create a noticeable space between words. In this code, we’re printing multiple values from the table. We do this by using the .. operator.

Looping through tables

```
local names = {"Andrew ", "Alfred ", "Alex ", "Amber ", "Alice "}

for i = 1,5 do
	print(names[i])
end
```

In this code, we’ve made a for loop. If you remember what this type of loop is, and how it works, then this concept will be easy to understand. If you forget how a for loop works, please reference your lesson four packet. Hopefully, you all remember how this type of loop works, but if you don’t, it’s okay.

For those of you that remember how a for loop works, you’ll be able to easily understand this. We have the exact same tables as we have above, with the same names, values, and everything. After the table is created, we have the for loop that we’ve been talking about. This particular loop is going to repeat itself five times. The variable i is always going to be equal to the number of re-

peats that the loop has gone through so far (If the loop is on its fourth loop, then i will be equal to four). Now, we're simply printing names[i] every time that the loop repeats. Knowing that the loop is going to repeat itself five times, we also know that i, in the end, will be equal to five. Each time the loop repeats, the next value in the table will be printed. The above code, if made correctly, will return:

Andrew
Alfred
Alex
Amber
Alice

Notice how a new line is created with each loop. This is due to the fact that print() creates a new line every time it is run. If you were to use term.write() instead, then all the names would be printed on a single line.

Tables within tables

```
local Table = {"Lewis", "Austin", {"Jim", “Kyle"}}
print(Table[1].." and "..Table[3][1].." are brothers.”)
```

Above, we have a table named Table. Within this table, we first have two strings, or values. The third value, as you can see, is a bit different. We have the french brackets again, which means that we are actually creating an entirely new table within the table. We don't have to give this new table a name, as it technically is just a value in the other table. It's okay, for now, if this concept is a bit confusing.

Within the new table that we've created within a table, you can see that it is, for the most part, the same. We create new values in the same way (separated by commas). At the end of this second table, we use a closing bracket. If we were to have more values after the second table, then we would simply put another comma, then make the next value. In this case, we do not need to add another value, so we have another closing bracket. Both of the tables are now created, so we can move on to referencing the material that they contain.

```
print(Table[1].." and "..Table[3][1].." are brothers.”)
```

In the line of code above, we are, once again, using the print() function. First, we print the first value within the first table (So Lewis will be printed). After this, we are printing the text and, which is just some text that we're printing. After this, things begin to get a bit confusing. First we are referencing the table named Tabled, which, in this case, is the only table that should be in your docu-

ment. Next, we're referencing the third value within that table. The third value, though, is another table, so what does this mean? Well, that's why there is a second number after 3. We are referencing the first value within the the third value. To better understand this, we'll just say that we're referencing table two, then referencing the first value in that table. After this, we're justing printing some more text to create a sentence. If done correctly, the above code will result in the following being printed:

Lewis and Jim are brothers.

If you wanted, you could also reference more than one tables-within-tables. The code would get quite confusing very quickly, but you could, technically, referencing a table within a table, then another table within that table, then another table within another table. Be warned, though, that the code will get confusing, and errors are likely.

Text-Based GUI

Simple response system

Let's create a simple input and response system. For those of you wondering what this is, think of a program like "Siri" (Made by Apple), only instead of talking to the program with your voice, you type. Take a look at the sample code below:

```
function lightGray(str)
	term.setTextColor(colors.lightGray)
	print(str)
end

function wgray(str)
	term.setTextColor(colors.gray)
	write(str)
end

function pos(xpos,ypos)
	term.setCursorPos(xpos,ypos)
end

function exit()
	term.setBackgroundColor(colors.black)
	term.clear()
	term.setCursorPos(1,1)
end

term.setBackgroundColor(colors.white)
term.clear()
pos(2,2)
lightGray("Enter text...")

while true do
pos(2,4)
wgray("-> ")
command = read()

if command == "hi" or command == "Hi" then
	print("\n Hi!")
elseif command == "hello" or command == "Hello"
then
	print("\n Hello.")
elseif command == "exit" or command == "Exit" then
	exit()
	break
```

```
end

term.setCursorPos(1,4)
term.clearLine()
end
```

We've seen codes much like this one before, but as you can see, we've used multiple things that we've learned in the past few lessons. In this code, we use our apis that we created last week (with some changes). We've also used the loops that we learned about in lesson four.

As for exactly what we'd be using this code for, you'll have to think of that yourself. You can simply leave it as a text input and output program (Where a user enters "Hi" and the programs responds "Hi"), or you can change it so that the user can enter specific commands to have the computer (emulator) do things. For example, you could have a command to shutdown or reboot the computer, where the user would enter reboot or shutdown, and the computer would respond accordingly. You could also run programs with this setup by using shell.run().

An Extra Tip

Numbers in tables

```
local numbers = {1, 2, 3}
print(numbers[2])
```

You can have numbers in tables as well. Overall, numbers in tables work the same way as text (strings) in tables. You reference them exactly the same, and you "make a list of them" the same way. The difference is when you are creating the table itself. Instead of putting the numbers in quotation marks, you simply need to put the numbers themselves, and that's all. Variables act much the same, as you should recall. You can use numbers in tables in much the same way as text in tables, and, of course, you can also accomplish math with them as well. To do this, look at the example below:

```
local names = {1,2,3}
print(names[2]+names[3])
```

Review

REVIEW 7.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

Which *best* describes a table?

- **A.** A special variable
- **B.** A set of containers each holding a different value that can be referenced individually
- **C.** A single container that holds a value

Check Answer

Lua Part II

- More with tables
- Command Arguments
- io.read()
- Math.random()

SECTION 1

More With Tables

In our lesson today and in lessons that we will have in the near future, you will notice that tables will continue to make a very large appearance. In this lesson, we're only using tables for a few things, but in the coming lessons, you should expect to see many tables being used. If you're all-set with tables, then you should be fine with the coming lessons. If you are at all confused with tables, please ask questions to try to get rid of any and all confusion.

Another way of making tables

Look at the program below:

```
local Table = {"Hi", "Hello"}
local Table2 = {Test = "Hi", Hi = "Hello"}

print(Table[1])
print(Table[2])

print(Table2["Test"])
print(Table2[“Hi"])
```

Above, there are two tables being named, one with the name Table and one with the name Table2. The first table has the normal table format that we learned about last lesson. Table2, on the other hand, has a slightly different format. In Table2, you can see that there are still two keys, but instead of having them referred to as [1] and [2], we're giving the keys names. The first key has a name of Test, and the second has a name of Hi.

When we reference information from the first table, we can use Table[1] and Table[2], like we did in the last lesson. For the second table, we use the names of the keys in place of numbers, so that we use Table2[“Test”] and Table2[“Hi”]. Notice how there are quotation marks around the keys. When we print the information of the two tables by using the two different methods, you will notice that they both return the same text. You can use either style of table (or array) in your coding, but it is up to you to decide wether you like naming the keys or just referencing them by number. Either way you choose, it will work if done correctly.

Command Arguments

Getting multiple pieces of information in one command

Command arguments are, put simply, different pieces of information that you give to a program. The program then looks at these pieces of information that you give it and performs actions based upon what exactly the information is. Arguments are one of the most useful things in many coding languages, and Lua is no exception.

To learn how to use command arguments (and how to make your programs work with them), please create a new program under the title of say. While the exact name of the program is not highly important, it may help in learning how to use arguments.

```
local Args = {...}
local text = Args[1]
local person = Args[2]

local function red(str)
    term.setTextColor(colors.red)
    print(str)
end

if Args[1] == nil then
    red("Correct Usage: say [text] [person]")
else
    if Args[2] == nil then
        red("Correct Usage: say [text] [person]")
    else
        print(person.." says "..text)
    end
end
```

The above code contains many things that we have yet to learn. Let's start at the very top of the code and work our way down. At the top of the code, you can see that we are making a table by the name of Args. The name for this table is not all that important, but you should know and be aware of the fact that it is in reference to arguments. Within this table, we have only three periods instead of keys, or strings (or numbers). What these three periods mean, when they are within the brackets, that is, is that the table's values will be whatever that arguments of the command are. To better explain this, let's take a look at the text below:

say hello Ryan

Using the code one the previous page, the above command (once it's entered) will result in the following text being produced on the screen:

```
Ryan says hello
```

If we look back at the command we entered, it is easy to see the parts of the command. Say is the name of the program that we are working with. Hello is the text that is going to be displayed, and since it is the first piece of information that we're giving the program itself (the name of the program is not a piece of information that the program itself uses), hello will be the first key within the table. This means that when we print Args[1], the text hello will be printed. Likewise, Ryan is going to be stored as the second key in the table, as it is the second piece of information that we're giving to the program. When we print Args[2], the text Ryan will appear.

Since the program now has all the information it needs, the parts of code below the table are now valid. We've made two variables, one called text and one called person. In think case, text is equal to whatever Args[1] is equal to, and person is equal to whatever Args[2] is equal to. We'll use these variables are just quick-access to the values of the table. Really, though, these variables are unneeded for the length of code that we're working with here.

After we declare those two variables, we have one of the APIs that we've been using in the last few lessons. The API we're using here is just changing the text color to red, and it rather simple. Since we use the API twice, even in this relatively short code, the API saves a couple of lines of code.

The conditional statement that we're using here is just to catch errors, for the most part. It simply checks to see if the user enters the correct amount of arguments when they enter the command for the program. If they have an error in their command, then a message telling them how to use the command correctly will be displayed in red. If they enter the command correctly, then the program will output with a response that is, technically, tailored to the exact information that the user is attempting to get.

There are obviously many things that you can now accomplish using command arguments. Instead of having to program the asking of questions to have a user enter information to complete a task, you can go right to completing a task. Command arguments will save time for both you and the users of your program(s). Here are a few program ideas that use command arguments:

Mad Libs (Stories), Calculators. Games, AI, Input-Output Programs

io.read()

We've already learned about how to read what a user inputs by using the read() function, but what if you want to do something a little bit different? Today, we're going to learn how you can have your program listen for a certain type of input — hitting enter. Look at the code below:

```
term.write("Enter to continue...")
io.read()
print("You hit enter!")
```

In this instance, the text Enter to continue… will appear on the screen, and then the program will wait for the user to hit enter before it continues. Once the user hits enter, the text You hit enter! will appear on the screen. Io.read() is a good way to create a one-step verification that the user is ready for a program to continue. Here are a few ideas for programs that use io.read():
Games, Tests, Q&A Programs, Startup Scripts, Verification for any program you can think of

Math.Random()

Generating a random number

Math.random() operates in exactly the way that you'd think it would. It generates a number. Take a look at the line of code below:

```
print(math.random(0,100))
```

In the above code, we're simply using the print function. Within this print function, we have something odd. At first glance, it would appear that we're just printing a variable. If it helps, then you can feel free to think of math.random() as exactly that. It is a variable with a variable of whatever number is generated. In this particular line of code, we're only looking for a number to be generated that is between the numbers of 0 to 100. As you can see, we have math.random(), then within the two parenthesis we have 0,100, which indicates that those are the two numbers that the random number must be between. You can have any two numbers you want here (For example, 0 and 100, 10 and 20, 1 and 1000000).

Math with random numbers

Using Math.random(), you can create randomly generated math problems, where you have two random (unknown) numbers being added together or multiplied. You can use any Lua-supported math operation.

```
local num1 = math.random(1,100)
local num2 = math.random(1,100)

print(num1)
print(num2)
print(num1+num2)
```

Above, we have two variables, num1 and num2 that are both equal to math.random(). Both of these variables will have different numbers, because math.random() generates a new number each time. Next, we're simply printing the values of num1 and num2, just so that we know what numbers have been generated before we do any math. Last, we have our mathematical operation. In this case, we're using addition. Supposing num1 is equal to 10 and num2 is equal to 15, then the above code will produce the following text:

```
10
15
35
```

Available (simple) math operations: +, -, *, /

Guess the number

The following program is just an example of a program that you can create using everything that we've learned so far.

```
--Presets
w,h = term.getSize()
number = math.random(1,100)
attempts = 1
maxattempts = 7
version = "1.0.1"
win = false

function thanks()
	term.setCursorPos(2,h-2)
	term.setTextColor(colors.gray)
	term.setBackgroundColor(colors.lightGray)
	term.clearLine()
	textutils.slowPrint("Thanks for playing!")
	term.setBackgroundColor(colors.gray)
	term.setCursorPos(1,h-1)
	term.clearLine()
	textutils.slowPrint("                    ")
	term.setBackgroundColor(colors.black)
	term.setCursorPos(1,h)
	term.clearLine()
end

--Introduction
	term.setBackgroundColor(colors.white)
	term.clear()

	term.setCursorPos(2,2)
	term.setTextColor(colors.gray)
	term.write("Guess the Number ")
	term.setTextColor(colors.lightGray)
	term.write("["..version.."]")
	term.setTextColor(colors.gray)

	term.setCursorPos(2,4)
	term.write("Press enter to continue...")
	io.read()

--Guess

while true do
	term.setCursorPos(1,6)
	term.clearLine()
```

```
    if attempts > maxattempts then
        break
    else
        term.setCursorPos(2,5)
        term.clearLine()
        term.setTextColor(colors.lightGray)
        term.write("Attempts  Remaining:  "..maxattempts-
attempts)
        term.setCursorPos(2,4)
        term.clearLine()
        term.setTextColor(colors.gray)
          term.write("Guess a number between 1 and 100:
")
        guess = read()
        guess = tonumber(guess)
            term.setCursorPos(2,7)
            term.clearLine()
        if guess == number then
            win = true
            break
        elseif guess == nil then
            term.setCursorPos(2,7)
            term.write("Please enter a number!")
        else
            if guess > number then
                term.setCursorPos(2,h-1)
                term.clearLine()
                        print("Your guess was too high. Guess
again.")
            elseif guess < number then
                term.setCursorPos(2,h-1)
                term.clearLine()
                        print("Your guess was too low. Guess
again.")
            else
                term.setCursorPos(2,h-1)
                term.clearLine()
                 print("An error has occurred.")
            end
            attempts = attempts + 1
            end
        end
    end

    --Finish
        if win == true then
            term.clear()
            term.setCursorPos(2,4)
            term.setTextColor(colors.lightGray)
```

```
        print("You win!")
        term.setCursorPos(2,6)
        print("You guessed the number in "..attempts.." attempts.")
        term.setCursorPos(2,8)
        print("You had "..maxattempts-attempts.." attempts remaining.")
        thanks()
    else
        term.clear()
        term.setCursorPos(2,2)
        print("Game Over")
        term.setCursorPos(2,4)
        term.setTextColor(colors.lightGray)
        print("The computer wins!")
        term.setCursorPos(2,6)
        print("The number was: "..number)
        thanks()
end
```

An Extra Tip

"Building" numbers

```
local num1 = 1
local num2 = 1
print(num1..num2)
```

You can build numbers with Lua. You're probably thinking that "building numbers" means adding them together, but in Lua, it doesn't! In the above code, we have two variables, one called num1 and the other called num2. Both of the variables are equal to one. Next, you see that we have a print() function. We first print num1, so 1, and then we use the .. (concatenation) operator to add num2 to the text. What this will do is not literal addition of numbers, but combining the numbers as if they were two words. This concept is easier to understand if you see what the above program produces as a result:

```
11
```

Review

REVIEW 8.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

When you use {...} in a table, what are you saying?

- **A.** That the table is filled with no values
- **B.** That the table is filled with the extra information given to a program upon a command being entered
- **C.** That the table is filled with values to be set from a read() function.

◀ Check Answer ▶

Lua Part III

- More with tables
- FS API (Config Files)
- Key-press GUI
- Minimalistic Design
- Account creation/Login programs

More With Tables

Creating sentences and iterating through tables

```
local Sentence = {"Hello! ", "This string ", "was read from ", "a table!"}
for i=1,#Sentence do
  term.write( Sentence[i] )
end
```

Hopefully, you've now all learned how to make tables and why they are useful within code. The above code, I believe, can now be explained with relative ease. As you can see, we have a local table named Sentence. When we would like to reference the table, we use the name, Sentence, to do so. Within the table, we have four strings. In this case, we're going to be creating a sentence with the table, so the strings within it do not make much sense on their own, but will combine later to form a readable text.

Now that the table is created, you can see that we're using a simple for loop, which we have used multiple times before. In this particular case, the for loop will repeat the line of code term.write(Sentence[i]) for a certain number of times. Remember that i is going to be equal to the numbers of times that the loop has repeated, so using Sentence[i] will produce the string of the table that has the same key, or location within the table, as the value of i. With normal for loops, there are two numbers (Such as 1 and 4) to indicate how many times that the loop should repeat. For this loop, we use 1 and #Sentence to indicate how many times the loop should repeat. This is just a simple way of telling the program to have that loop to repeat for the same amount of times that there are strings in the table. In this case, the loop will repeat itself four times.

Hopefully, this addition to your knowledge of tables is simple to understand. The lessons have been designed to make learning tables as easy and struggle free as possible.

SECTION 2

FS API

What is the FS API?

The FS (Filesystem) API is a set of useful functions that lets you make, edit, and work with the files on your computer (emulator). The FS API lets you have absolute control over your computer. Using it, you can create programs that are far more useful than anything we've discussed or created thus far.

In this lesson, we'll be using the FS API to do multiple tasks, such as creating folders and files, writing to files, deleting files, checking if a file or folder (directory) exists, and viewing the size (in bytes) of a file, along with other various operations that the FS API can do. By the end of this lesson, you'll have a basic understanding of how to create and write to files and read the text of those files. If you're confused now, everything will be explained soon.

Creating and editing files

With the FS API, you have the ability to create new files from within a program, meaning that you can use one program to create many different files. To do this, we use a number of different lines of code. Take a look at the code below:

```
testprogram = fs.open("test", "w")
testprogram.write("Hello!")
testprogram.close()
```

Once you understand what the lines of code mean, the above code becomes remarkably easy to understand. There is a variable named testprogram, which is just a name to indicate what we're working with. This variable is equal to fs.open("test", "w"). Of course, without explanation, this function we're using makes little to no sense. Hopefully, you can infer was fs.open() in itself does. The function simply creates a new file or edits an existing one. Within the two parenthesis, you see that we have to strings, test and w. The first string in the parenthesis will be the name of the file being created or edited, and the second string is a file handle. A file handle is basically just a letter that tells the program what exactly you're going to be doing with the file. There are four types of file handles:

r - Opens a file, like opening a program with the edit command, and only allows the file to be read, not edited.

w - Opens a file and removes any and all existing data.
a - Opens a file and allows the editing of the file while keeping existing data.
b - Opens a file for binary access instead of text access.

Now that we know what a file handle is, it is a little bit easier to understand exactly what the code above does. Basically, the program will create a new file named test, and if that file already exists, then all data within that file will be removed and replaced with what we're about to put in it.

The next line, testprogram.write("Hello!"), should be somewhat self-explanatory. Since testprogram is equal to fs.open("test", "w"), we can use testprogram in places that we would normally use fs. You can think of this as a slight shortcut, in a way. Since we know what write() does, you should all know what the rest of this line of the code does. The test file is going have the text Hello! written to it. This line of code has the same effect as you editing the file and simply typing Hello!, but in this case, you never need to touch the file itself.

Lastly, we use testprogram.close() to indicate that we're done working with the file. This line of code will have the effect as you editing the program, using control and the arrow keys to save the file, the manually closing the file. If you do not close the file, the it will not be saved.

When you create a new file, the name of the file will be whatever you use as the first string in the fs.open() function. The location of that file will ignore the current directory you're in, meaning that if you just have one word, the file will be create with that word as its name in the main directory. If you're doing most of your work within a folder, you must put the name of the folder and a / before the name so that the file is created within that folder. Using the same method, you can create new folders and place the new files within them if you so choose.

Unfortunately, not all parts of Lua or even ComputerCraft are available with the emulator which we are working with, as we have seen, and some of the things relating to the FS API are included in the list of unsupported features. The code we discussed above works perfectly well, but if you attempt to use a rather than w as the file handle, the program runs into issues. Eventually, a fix for this may be provided.

Writing variable information to a file

Even without some of the normal features, the FS API still has an extensive list of things that it can help you and your codes accomplish. We're now going to be tak-

ing a look at a program that will create a file and write variables and strings to it. We're next going to make a program that will print out the values of the variables and will be able to retrieve the strings.

To start, look and copy down the following code:

```
information = fs.open("data", "w")
information.write("num1 = 5\n")
information.write("num2 = 8\n")
information.write("text1 = 'Hello!'")
information.close()
```

Using what we learned above, we know basically what this code will do. This code, when run, will create a file called data and will have the information we've put in the information.write() function. For the most part, if not entirely, you should have a relatively good or okay understanding of this. Remember, \n will create a new line, and in this case, that new line will end up putting all the variables we're making on a different line of the new file. Speaking of variables, you can see that in the above code, we are, in fact, writing variables and their corresponding data to the file. Since the file being create will simply have exactly what we write in it, we will be able to look at, find, and reference those variables.

```
shell.run("../../../data")

print(num1)
print(num2)
print(num1+num2)

print("\n"..text1)
```

To understand the above code, you have to completely realize that the previous code created a file and that we're now trying to get the information and data that belongs to the variables we created. The first list is just to make sure that the file that was created has been run at least once (so that we're able to use the variables), then the rest is just simple printing and adding using the values and strings of the variables. The most confusing part of this, I believe, is the original creation of the new file. From then on, it's mostly just functions that we've used before.

Key-Press GUI

Let's now take a quick break from the FS API and move onto something a bit more in line with what the next and last lesson is going to be mostly about. Today, we're going to be creating a key-press GUI (Graphical User Interface). A key-press GUI, at least the way we're going to make it, is a program that relies on users hitting keys on their keyboard to navigate and interact with the program. Take a look at the code below:

```
-- Presets
local version = "1.0"

function clear()
 term.clear()
 term.setCursorPos(1,1)
end

function help()
      print("Welcome to Listener "..version)
      print("Press A to clear the screen")
      print("Press Q to leave")
end

clear()
help()

while true do
 event, id = os.pullEvent("char")
  if id == "t" then
      print("You pressed "..id..".")

  elseif id == "h" then
      print("You pressed "..id..".")

  elseif id == "c" then
      print("You pressed "..id..".")

  elseif id == "a" then
   clear()
   help()

  elseif id == "q" then
      clear()
      break
  end
end
```

Hopefully, the above code, for the most part, makes sense. There are part of it that we've net yet discussed what exactly they do, but most parts of it simply use things that we've now known for many lessons now. Being that the majority of the program can be explained and understood by notes from previous lessons, I'll only be covering a relatively specific set of information.

As you can see, we have a while, or a continuous loop. With that loop, we have two variables -- event and id -- that are both equal to OS.pullevent("char"). In short, this means that the computer will listen for an event in which the user presses a key on the keyboard. Now that we've told the computer what event it is listening for, we use a conditional statement to detect which character the user presses. Depending upon the character that is pressed, different information will be displayed.

In the example above, we're just displaying simple text onto the screen, but this code could be enhanced by using different layouts, colors, and functions. Depending upon the program that you're creating, a key-press GUI could greatly enhance the user experience. Here are just a couple of ideas for programs that you could use key-press GUIs for:

Calculators. Games, File Browsers, Text Editors, Guess a number between 1 and 10, Quizzes, Tests, Any program with keyboard shortcuts, ect

What can you think of?

Once we get to the end of this class, I would like all of you to either work by yourself or with a partner and begin work on some sort of program that uses key-press GUIs. You can make a program like one from the list above, or a completely different idea. Feel free to incorporate as many other things that you know about Lua into your code. Obviously, this is more of a way to check your understanding and to show other members of the program what you know, so this project will not graded. The only assessment being truly given will be based on how creative you are (but remember, you will not be graded at all).

If you finish you project today, you may present to your peers at the end if class. For those of you who do not have the ability to present today, for whatever reason, you will be able to present at the end of the next lesson and at the end of the very first Java lesson. If you and your partner do not wish to present, then that is okay as well. Hopefully, though, everyone will be willing to present. Good luck! Let's see what you all know

about the coding language Lua through the medium of various codes and programs!

SECTION 4

Minimalist GUI

What is a minimalistic GUI?

For this section of the lesson, let's talk about what exactly a minimalistic GUI is. In fact, let's first talk about what exactly a GUI is. A GUI is basically just the way that a program looks and how it responds to a user's actions. It is an abbreviation for Graphical User Interface, which, as we [hopefully] know, means the way it looks and how the user interacts with different parts of the program.

Now that we know what a GUI is, we can guess what a minimalistic GUI is. Let's have a quick open discussion about what exactly this means. If you want, you may take notes in the space below or you can just listen.

Is a minimalistic GUI a "good" GUI?

To answer the above question, we first need to understand exactly what is classified as a "good" GUI. A good GUI is a GUI that works. If the program you have is useable, accomplishes the task intended for it to accomplish, and is understood by the user, the the GUI is good. We also need to know what a "bad" GUI is. A bad GUI is a GUI that does not work, fails to accomplish the intended task, or confuses the user.

Now that we know what both of those are classified as, we can answer the above question. A minimalist GUI is good as long as it is able to work correctly and performs as intended. If the user is confused due to the GUI, then the GUI, either minimalist if or not, is bad. A GUI can also be tested based upon how much information is presented. If there is too much information that causes the user to become confused, then there are problems presented. Likewise, if there is too little information, a user does not get exactly what they may be looking for in your program.

So, knowing what was talked about above, do you think minimalism is a good thing when it comes to GUI or a bad thing? You may think that that question is unanswerable as in order to judge GUI, you must be able to use it. Each program has a type of GUI that works best

for it, and each program has a different purpose or intention.

This section of this lesson was mostly just to get you all thinking, and to introduce important points of the last lesson. Hopefully, you'll all be able to create create GUIs for all your programs!

FS API (Continued)

Account creation and login program

Here is an example of a login program that uses the FS API to store user data and has customized settings.

Setup File

```
local w,h = term.getSize()

term.setBackgroundColor(colors.white)
term.setTextColor(colors.lightGray)
term.clear()

term.setCursorPos(1,2)
write(" Username: ")
username = read()
write(" Password: ")
password = read("*")
write(" Max Attempts: ")
max = read()
write(" Background Color: ")
backcolor = read()
write(" Text Color: ")
textcolor = read()


account = fs.open("Lesson9Additions/users", "w")
account.write("User = '"..username.."'")
account.write("\nPass = '"..password.."'")
account.write("\nmaxAttempts = "..max.."")

account.write("\nbg = "..backcolor.."")
account.write("\ntc = "..textcolor.."")
account.close()

term.setBackgroundColor(colors.lightBlue)
term.setTextColor(colors.white)
term.setCursorPos(1,h)
term.clearLine()
term.setCursorPos(1,h-2)
term.clearLine()
term.setCursorPos(1,h-1)
term.clearLine()
textutils.slowPrint(" Setup complete")

sleep(2)
```

```
term.setCursorPos(1,1)
term.setBackgroundColor(colors.black)
term.clear()
```

Login File

```
shell.run("users")
local w,h = term.getSize()

for i = 1,maxAttempts do
term.setBackgroundColor(colors.white)
term.setTextColor(colors.lightGray)
term.clear()

term.setCursorPos(1,2)
write(" Username: ")
name = read()

if name == User then
	write(" Password: ")
	loginPass = read("*")

	if loginPass == Pass then
		term.setCursorPos(1,1)
		term.setBackgroundColor(bg)
		term.setTextColor(tc)
		term.clear()

		term.setCursorPos(2,2)
		print("Welcome, "..User)

sleep(1)
term.setBackgroundColor(colors.lightBlue)
term.setTextColor(colors.white)
term.setCursorPos(1,h)
term.clearLine()
term.setCursorPos(1,h-2)
term.clearLine()
term.setCursorPos(1,h-1)
term.clearLine()
textutils.slowPrint("                ")
		break
	else
		print("Incorrect Password")
	end

else
	print("Incorrect Username")
end
end
```

Information-Getting File

```
shell.run(“users")
print("User Information:")
print("Usernane: "..User)
print("Password: "..Pass)
print("\nGUI Information:")
print("Background Color "..bg)
print("Text Color “..tc)
```

Generated File

```
User = 'Stephen'
Pass = 'K'
maxAttempts = 1
bg = colors.white
tc = colors.lightGray
```

An Extra Tip

Better APIs

```
function Red(str)
    term.setTextColor(colors.red)
    print(str)
end
```

You know how when you use term.setBackgroundColor, you have to use the term. before the other text? In the APIs that we've created in prior lessons, we used shell.run("file name") to indicate where we were getting our APIs from. One thing you may have noticed is that you did not have to use a special abbreviations in front of the functions when you used it. In many cases, this technique is fine and works quite well, but there are some cases where using this technique is inappropriate or simply not professional. The correct way of referencing APIs you're using for your programs is as shown below:

```
os.loadAPI("file name")
```

In this case, the file name is the name of the file in the root directory that has your API's functions within it. This is the same as using shell.run(), but is more professional. To reference one of the functions, you would use something along the lines of test.cPrint(), supposing the file name is test and the function you're using is cPrint().

Review

REVIEW 9.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

What do you use to quickly find the number of keys in a table?

- **A.** @Tablename
- **B.** #Tablename
- **C.** *Tablename
- **D.** &Tablename

◀ Check Answer ▶

Lua Part III

- GUI Design
- Mouse-Based GUI
- Combining Key-Press and Mouse-Based GUIs
- “Animations”
- Pastebin and “Donating” your programs

SECTION 1

GUI Design

When creating your programs, you always want to think about how the programs will look. The look of a program can greatly affect the way that the user interacts with the information within the program. The right look for your program is up to you, but here are a couple of ideas to get your thinking:

Simple layouts

Most programs consist of a simple, yet effective layout. They have a header which describes what the information below it will be about, and, of course, they have the information and text itself which tells the user about exactly what they are looking for within your program. Compared to other types of layouts, this style is easy to create, not very difficult to understand, and does a fine job at what it is meant to do. With each layout, though, there are different jobs that the layout may need to do. For example, you may want your program's layout to be very simple, maybe even without a header, until a user click on a certain button that will cause menus and other information to appear. This type of layout is more interactive, and is, at first, more content-focused. Once the user clicks on the area that causes the other information to appear, the program's layout becomes more focused on transitional information, or information that is meant to be a transition into another part of the program. Even with these hidden menus, you can still call this type of layout a simple layout.

Advanced layouts

Of course, you also have more advanced layouts, which incorporate multiple parts of design that come together to form a unique and elegant experience. A perfect example of an advanced layout is one that uses hidden menus and is effectively responsive to the user's actions. An effect response is one such as customized background colors and different information being displayed depending upon the user's preferences.

The experiences

In most programs, the overall GUI is not nearly as important as the experience itself. In order to have a good experience, the GUI must have a good amount of though put into it, but it is a combination of multiple things that create a great user experiences. To enhance

the experience of your programs, you may want to incorporate animations, which we'll talk about later, custom colors, functionality, and content. To create a truly great experience, you must find a balance between all of these that will not only give the user what they want, but make them happy at the same time.

In this lesson, we'll be learning how to improve the user experience by using animations coupled with a clickable GUI. Using both of these, you have the ability to create a program that will be orders of magnitude better than programs that you've created in the past. Keep in mind, this is our last Lua lesson, and at the end we'll be presenting some projects we've made. If you want, take what you learn today and apply it to your project, then share it with us at the end of the first lesson of Java.

Mouse-Based GUI

Detecting clicking

```
num = 0

term.clear()
term.setBackgroundColor(colors.lightBlue)
term.setCursorPos(2,2)
term.write(" ")

while true do
    event, button, x, y =os.pullEvent("mouse_click")
    if x == nil or y == nil then
        term.setBackgroundColor(colors.blue)
        term.setCursorPos(3,2)
        print(“Something went wrong.”)
    end
    if x == 2 and y == 2 then
        num = num+1
        term.setBackgroundColor(colors.blue)
        term.setCursorPos(3,2)
        if num > 1 then
            print(" You've clicked the button "..num.." times")
        else
            print(" You've clicked the button "..num.." time")
        end
    end
end
```

In the above code, we first have a variable, num, which is equal to 0. This variable is just going to serve as a place holder, which you'll see exactly what it does in a moment. Next, we have simple positions and colors that will just create a nice(ish) look to the program. Notice how we're going to have a light blue space appearing in the location of (2,2). So far, this code should make sense. The next part can get a bit confusing.

After we set the colors and the position, we have a while loop, or an infinite loop. Within this loop, we have the variables event, button, x, and y equal to os.pullEvent(“mouse_click”). This is basically just telling the computer that it should listen for the event of a mouse button being clicked at an x,y location. From this, you can tell that the event variable is just specifying the event, button is specifying the button of the mouse, and x,y are coordinates.

Now, we can basically just use the x and the y variables to create different parts of a conditional statement. If the x coordinate or the y coordinate are nil, or basically non-existent, then clearly, the user did something wrong or the program interpreted the data incorrectly. Either way, something did not go to plan, so we're just going to print out that something went wrong. Next, we have a conditional statement that is detecting if both x and y are equal to 2. If they are, then we're going to add one to num, then print text depending upon what number num is. If num is one, then the program knows to have the word "time(s)" be singular, and if num is greater than one, then the program knows to use the plural version.

Remember, in the beginning of the program we created a space of light blue in the location (2,2). In the conditional statements, we're printing in the location of (3,2), meaning that the text will be displayed in the space after the button. Also keep in mind that we are listening for the user to click on the location (2,2), so in a sense, we created a button.

This is a very simple version of listening for a user to click a location on the screen, but the effects of it are the same. With this, you're able to create interactive programs in which a user must click on different parts of the UI in order to do certain tasks. Depending upon what program you're making, a mouse-based GUI can greatly improved the experience.

Multiple lines and spaces being detected

```
num = 0


term.clear()
term.setBackgroundColor(colors.lime)
term.setCursorPos(2,2)
term.write("    ")
term.setCursorPos(2,3)
term.write("    ")

while true do
    event, button, x, y =os.pullEvent("mouse_click")
    if x == nil or y == nil then
        term.setBackgroundColor(colors.green)
        term.setCursorPos(5,2)
        print("Something went wrong.")
    end
    if x>1 and x<5 and y>1 and y<4 then
        num = num+1
```

```
            term.setBackgroundColor(colors.green)
            term.setCursorPos(5,2)
            if num > 1 then
                print(" You've clicked the button "..num.." times")
            else
                print(" You've clicked the button "..num.." time")
            end
        end
    end
```

The above program is much like the one we made before, but with a little twist. Overall, we're doing exactly the same thing, only we're creating a multi-line and multi-space button that is clickable across the entire extent of the button. Most likely, you're thinking that this is rather easy, which it is, but you have to know how to use inequalities (roughly). You also have to know that > and < do not include the numbers that they are referencing. In math, this can be explained by x>1, meaning that x can be 2, 3, 4, 5, and so on for an infinite amount of numbers, but none of the numbers are less than or equal to one. In Lua, you have to position and detect stuff much like this. Above, when we say if x>1 and x<5 and y>1 and y<4, we're looking for an x value of 2,3, or 4, and a y value of 2 or 3.

Hopefully, this makes sense. If it doesn't, just go over the code and you'll most likely be able to understand it. In addition to knowing the above, remember that to stop an infinite loop you must use the break command. In this case, you could put break in the first conditional statement and the program will simply stop. Depending on the type of program you have, you will always have a break command that is able to activated by the user somehow. Using the button example, you may want to create a red button to indicate that it will exit the program, then when the user clicks the red button, the while loop breaks.

Using os.pullEvent and detecting mouse clicks, you can create programs that are completely reliable on the user clicking on individual parts of a program. Using a conditional statement, you can have a multitude of different actions and tasks being carried out by the simple click of a mouse (or tap on a trackpad). These is, quite literally, nearly no limit to what you can do with this. You can run programs, create files, add and subtract to/from variables, create or add to tables, and so much more.

SECTION 3

Animations

Animations are a vital part of almost all programs that we use today. When you minimize a window, an animation takes place that puts the window into the dock. When you load a page in a web browser, there is (normally) an indicator of how much the page has loaded so far. In a sense, animations can be seen in almost every part of most operating systems. Today, we're going to attempt to replicate the animations the we see. First, we're going to start with some simple animations where we have a loading screen, a drop-down menu, or something of that kind. The animations we create won't necessarily be like the animations we see on computers, but they'll be close.

```
num = 0

function once()
    term.write(" ")
    sleep(0.00)
    term.write("Y")
    sleep(0.00)
    term.write("o")
    sleep(0.00)
    term.write("u")
    sleep(0.00)
    term.write("'")
    sleep(0.00)
    term.write("v")
    sleep(0.00)
    term.write("e")
    sleep(0.00)
    term.write(" ")
    sleep(0.00)
    term.write("c")
    sleep(0.00)
    term.write("l")
    sleep(0.00)
    term.write("i")
    sleep(0.00)
    term.write("c")
    sleep(0.00)
    term.write("k")
    sleep(0.00)
    term.write("e")
    sleep(0.00)
    term.write("d")
```

```
sleep(0.00)
term.write(" ")
sleep(0.00)
term.write("t")
sleep(0.00)
term.write("h")
sleep(0.00)
term.write("e")
sleep(0.00)
term.write(" ")
sleep(0.00)
term.write("b")
sleep(0.00)
term.write("u")
sleep(0.00)
term.write("t")
sleep(0.00)
term.write("t")
sleep(0.00)
term.write("o")
sleep(0.00)
term.write("n")
sleep(0.00)
term.write(" ")
sleep(0.00)
```

```
        term.write(num)
        sleep(0.00)
        term.write(" ")
        sleep(0.00)
        term.write("t")
        sleep(0.00)
        term.write("i")
        sleep(0.00)
        term.write("m")
        sleep(0.00)
        term.write("e")
        sleep(0.00)
        term.write(" ")
        sleep(0.00)
end

function moreThanOnce()
        term.write(" ")
        sleep(0.00)
        term.write("Y")
        sleep(0.00)
        term.write("o")
        sleep(0.00)
        term.write("u")
        sleep(0.00)
```

```
term.write("'")
sleep(0.00)
term.write("v")
sleep(0.00)
term.write("e")
sleep(0.00)
term.write(" ")
sleep(0.00)
term.write("c")
sleep(0.00)
term.write("l")
sleep(0.00)
term.write("i")
sleep(0.00)
term.write("c")
sleep(0.00)
term.write("k")
sleep(0.00)
term.write("e")
sleep(0.00)
term.write("d")
sleep(0.00)
term.write(" ")
sleep(0.00)
term.write("t")
```

```
sleep(0.00)
term.write("h")
sleep(0.00)
term.write("e")
sleep(0.00)
term.write(" ")
sleep(0.00)
term.write("b")
sleep(0.00)
term.write("u")
sleep(0.00)
term.write("t")
sleep(0.00)
term.write("o")
sleep(0.00)
term.write("n")
sleep(0.00)
term.write(" ")
sleep(0.00)
term.write(num)
sleep(0.00)
term.write(" ")
sleep(0.00)
```

```
        term.write("t")
        sleep(0.00)
        term.write("i")
        sleep(0.00)
        term.write("m")
        sleep(0.00)
        term.write("e")
        sleep(0.00)
        term.write("s")
        sleep(0.00)
        term.write(" ")
        sleep(0.00)
end

function draw()
term.clear()
term.setBackgroundColor(colors.lightBlue)
term.setCursorPos(2,2)
term.write(" ")
end

draw()

while true do
      event, button, x, y =os.pullEvent("mouse_click")
      if x == nil or y == nil then
            term.setBackgroundColor(colors.blue)
            term.setCursorPos(3,2)
            print("Something went wrong.")
      end
      if x == 2 and y == 2 then
            num = num+1
            term.setBackgroundColor(colors.black)
            term.clear()
            draw()
            term.setBackgroundColor(colors.blue)
            term.setCursorPos(3,2)
            if num > 1 then
                  moreThanOnce()
            else
                  once()
            end
      end
end
```

In the above program, we're using buttons and functions, along with the sleep function, to create a very quick animation. Remember that a sleep of 0.00 results in a very quick sleep. When you run this file, you can see that the animation that takes place offers a nice-looking and smooth display of text.

Drop-down menu

In many programs, when you click on a certain area, a drop-down menu appears from which you're able to select different options. We're going to create a relatively basic drop-down menu which uses mouse-clicks and animations. We'll be taking a look at a program that makes use of multiple while loops.

```
local w,h = term.getSize()

function userMenu()
	term.setBackgroundColor(colors.gray)
	term.setTextColor(colors.white)
	for i = 1,h-4 do
		term.setCursorPos(w-10,i)
		print("          ")
		sleep(0.00)
	end

	term.setCursorPos(w-9,2)
	print("Reboot")
	sleep(0.00)
	term.setCursorPos(w-9,4)
	print("Shutdown")
	sleep(0.00)
		term.setCursorPos(w,h-4)
		term.setBackgroundColor(colors.red)
		print(" ")

	while true do
		term.setTextColor(colors.white)
		event, button, x, y
=os.pullEvent("mouse_click")
		if x == nil or y == nil then
			cPrint("What")
		end
		if x == w and y == h-4 then
			break
		elseif x>w-11 and x<w+1 and y == 2 then
			os.reboot()
		elseif x>w-11 and x<w+1 and y == 4 then
			os.shutdown()
		end
	end
end
function draw()
term.clear()
term.setBackgroundColor(colors.lightBlue)
term.setCursorPos(2,2)
```

```
term.write(" ")
term.setBackgroundColor(colors.black)
end

while true do
	draw()
	event, button, x, y =os.pullEvent("mouse_click")
	if x == nil or y == nil then
		print("Something went wrong.")
	end
	if x == 2 and y == 2 then
		userMenu()
		draw()
	end
end
```

Above, we are, once again, using buttons. In this case, clicking the button triggers and animation of a drop-down menu “dropping-down”. Once the menu is displayed, the user has the option to click one of the options of to click the red button to go back to the original button. This switching back to the normal button is done by breaking one while loop while another while loop is still running, so that even though one of the while loops is broken, the other continues. You can think of this much like conditional statements within conditional statements, only replacing the conditional statements with while loops.

SECTION 4

Pastebin and “Donating” your programs

If you want, you have the option to either put your program codes on to paste paste bin by using the command:

pastebin put filename

This will upload your code to the internet where it will be viewable by the world. This is good if you ever think you’ll want to show someone your work.

A Few Extra Tips

Errors

error(“Something went wrong”)

Using the above function, you can create custom error messages. This will cause the program to crash and display the custom message that is within the parenthesis.

Hello World

```
local colors = {colors.blue, colors.green, colors.red, colors.yellow, colors.black, colors.white}

term.setTextColor(colors[math.random(1,6)])
textutils.slowPrint("Hello World!")
```

The above program will slowly print (In alternating colors) Hello World.

Custom Termination

```
while true do

term.setBackgroundColor(colors.white)
term.clear()
term.setCursorPos(2,2)
term.setTextColor(colors.lightGray)
print("This is a test")

local event = os.pullEventRaw()
  if event == "terminate" then
       programName = shell.getRunningProgram()
      term.setBackgroundColor(colors.black)
      term.clear()
      term.setCursorPos(1,1)
          error("User has terminated the program: "..pro-
gramName,0)
      break
  end
end
```

With the above code, you are able to create a custom termination message. You can change the text with the error() function to display a different message when the user terminates the program.

Review

REVIEW 10.1 Lorem Ipsum dolor amet, consectetur

Question 1 of 5

What is the most important part of GUI design?

- **A.** The layout
- **B.** The colors
- **C.** The text
- ✓ **D.** A combination of all of the above

◀ Check Answer ▶

More Coming Soon...

--

The commenting symbol for Lua. Example: — This is a comment

Related Glossary Terms

Drag related terms here

Index

Find Term

..

The symbol for telling a program to create the next phrase, variable, or number after the prior phrase, variable, or number. Example: print(variable1..variable2)

Related Glossary Terms

Drag related terms here

Index Find Term

\N

The symbol to create a new line of printing within a single line of code. Example:
print(“Hello\nHow are you?”)

Related Glossary Terms

Drag related terms here

Index Find Term

==

The symbol for have the program check to see if a user's input is exactly equal to a number, variable, or phrase. Example: if input == "hello" then

---

Related Glossary Terms

Drag related terms here

Index Find Term

Else

The command for the program to do something if the options for if or elseif are not true. Example: else print(“That doesn’t work!”)

Related Glossary Terms

Drag related terms here

Index Find Term

Elseif

The command for the program to have a secondary option in a conditional statement.
Example: elseif input == “goodbye” then print(“Goodbye!”) end

---

Related Glossary Terms

Drag related terms here

Index Find Term

End

The command that tells a program that a certain park of code is finished being executed.

Related Glossary Terms

Drag related terms here

Index Find Term

Function

An operation or set or code that can be run multiple times, and can change or create text.

Related Glossary Terms

Drag related terms here

Index Find Term

If

The first part of a conditional statement.

The command for the program to check if this, then that. Example: if input == “hello” then

---

Related Glossary Terms

Drag related terms here

Index Find Term

Local

The attribute applied to a variable that tells the program that the variable is only for use within that single program, and no other programs can have access to it. Example: local var = “Hello”

Related Glossary Terms

Drag related terms here

Index Find Term

Print()

The basic function to print text on the screen. Example: print(“Hello World!”)

Related Glossary Terms

Drag related terms here

Index Find Term

Read()

The command for the program to see what a user's input is, and then store that information into a variable.

Related Glossary Terms

Drag related terms here

Index Find Term

Sleep()

The command for the program to wait a certain amount of time (in seconds) to wait before executing the next part of the code. Example: sleep(8)

Related Glossary Terms

Drag related terms here

Index Find Term

Term.clear()

The basic function to clear all information and text on the screen.

Related Glossary Terms

Drag related terms here

Index Find Term

term.clearLine()

The basic function to clear all information and text on a single line.

Related Glossary Terms

Drag related terms here

Index Find Term

Term.scroll()

The function to move all text and information on the screen up one line. Example:
term.scroll(8)

Related Glossary Terms

Drag related terms here

Index Find Term

term.setCursorPos()

The basic function to set the area, using an x and y grid, for where the next information or text will be displayed. Example: term.setCursorPos(1,1) or term.setCursorPos(x,y)

Related Glossary Terms

Drag related terms here

Index Find Term

Term.write()

Like print(), term.write() prints text. Unlike print, it does not create a new line after it.
Operates the same as print() in every other way.

Related Glossary Terms

Drag related terms here

Index Find Term

Then

In that case.

The command for the program to do an action once it has checked to see if something is true. Example: if input == “hello” then print(“Hello!”) end

---

Related Glossary Terms

Drag related terms here

Index Find Term

True

When the program checks to see if something equals something else, and it does, then the statement is true.

Related Glossary Terms

Drag related terms here

Index Find Term

Variables

Numbers: No quotation marks needed. Example: var = 1

Words (Strings): Quotation marks needed! Example: var = “Hello”

Related Glossary Terms

Drag related terms here

Index Find Term

Write()

The basic function to ask a user to enter text into a program. Example: write(“What is your name?”)

Related Glossary Terms

Drag related terms here

Index Find Term