14.3.1 Bash Scripting Logic

Click one of the buttons to take you to that part of the video.

Bash Scripting Logic 0:00-0:48

Let's talk about using control structures within shell scripts. A basic shell script typically executes straight through, from the beginning down through the end. This may work fine in some situations. But suppose we need the script to actually make some decisions. Based on user input, or maybe the output from a command, we, instead, may want the script to determine a different course of action, depending upon what that input was. This can be done by implementing control structures within the script. And there are several different types of control structures that we can use, and each one functions in a slightly different way. We're going to look at if-then-else, we're going to look at case, and then we're going to look at looping structures that you can use within a script.

The if/then/else/fi Construct 0:49-1:55

Let's begin by looking at the if-then-else structure. Using an if-then-else structure within your shell script gives your script the ability to execute different commands based upon whether a particular condition is true, or whether it's false. The structure is shown here. We enter 'if' followed by a condition. The if part of the structure tells the shell to determine whether or not this condition, right here, evaluates to true, or if it evaluates to false. If the condition evaluates to true, then this set of commands is run--the commands that appear under the "then" part of the structure. But if the condition evaluates to false, we execute the commands that fall under "else". This kind of structure can be really useful in situations where we need to branch based upon a user's input. It's also useful in situations where we need to validate the input that the end user has entered from the read prompt.

Input validation means we want to check and see whether the user entered a valid value at the prompt, or whether they entered something that's either incorrect or just doesn't match what it is we're looking for.

Input Validation 1:46-3:30

For example, let's suppose we've created a script, and the script will ask the end user to enter the name of a directory that he or she wants to add to the end of the PATH environment variable. When we add the directory to path, it would be, really, a good idea to run a quick test and just verify that the directory that the end user entered actually exists in the file system. If it does, then we should go ahead and add it to the PATH environment variable. If it does not exist, then we should post an error message on the screen, telling the user, "Hey, I can't find this directory. This is what this structure, here, does. In this example, we have our condition right here. This condition actually calls a utility called test. It tells test to check and see whether the directory contained in the newpath variable right here--which we are assuming we read in with the read command from the end user--it checks to see if that directory actually exists. And this is specified by the -d option. Essentially, this condition runs test -d followed by the path that the end user entered in at the read prompt in the script. And all the test command does is say, "Yep, it exists," or, "Nope, it doesn't exist." There, we have our true or false values, right? If it's true--if test runs and it finds this directory that the end user entered in-- then we run this set of commands. But if it doesn't exist, then we run the "else" part of the if-then-else structure; in which case, we just use the echo command followed by the name of the path that they entered. Then we add text, saying, basically, "Hey, this doesn't exist.

The test Command and Options 3:31-5:20

The test command we just looked at is extremely useful within an if-then-else structure. There are actually several other options that you can use with test within an if-then-else structure to perform other types of tests.

We already looked at dash -d. The dash -d option checks to see if a particular file exists. If it does, is it actually a file, or is it actually a directory? If it's a directory, it returns a value of true. If it's not a directory, or if it doesn't even exist at all, then we get a value of false.

-e is also another one that I use all the time. This checks to see if a file exists. Understand that the -e option doesn't differentiate between files or directories. All it does is look and see if there is an entity in the file system with the name that you specify. And if it's there, we return a value of true. If not, we return a value of false.

The -f option is kind of the counterpart of -d. It checks to see if the file that you are looking for exists. And, if it does, is it an actual file, and not a directory?

The -G option, capital G, checks to see if the file exists and whether or not that file is owned by a specific group.

The -h or -L--they're both the same--option with test will check to see if the file exists, and whether or not that file is a symbolic link.

The -O option checks to see if the file we specify exists and whether it is owned by a particular user account.

The -r option checks to see if the file exists and whether the read permission has been granted to that file.

-w checks to see if the file exists and whether or not the write permission has been granted.

-x checks to see if the file exists and whether or not the execute permission has been granted.

As you can see, you can use test to evaluate all kinds of conditions with respect to files and directories in the file system. Test is not limited to just seeing whether or not a file or directory exists.

Comparing Text and Numbers 5:17-6:11

You can also evaluate two different conditions with test.

For example, this command will check to see if text1, here, is the same as text2. Is it true, or is it false? You can also check to see whether these two text strings are not the same. Basically, it's just the opposite of the command we used before. We enter 'test' followed by the first text string, and then the exclamation point and an equals sign (!=), and then the second text string. In this case, if the two text strings are not the same, it will evaluate to true. If they are the same, it will evaluate to false. It's just, basically, the converse of the first example.

For example, we can use test to see if num1 equals num2, true or false? We can also test to see whether or not num1 is less than num2, or we can also test to see if num1 is greater than num2.

The case Construct 6:12-7:25

You can also use a related structure in your scripts called a case structure. The case structure is, really, just a glorified if-then-else statement. If-then-else works great if we have a particular condition that can be evaluated in one of two ways, either true or false. But if-then-else does not work so great if we need to evaluate a condition that could be evaluated in many different ways. If there are more than two ways that a condition could be evaluated, then you should use a case statement instead. The structure of a case statement is shown here. We start with 'case', and then we have a variable, whatever it happens to be. And then we say, "Okay, case." Then we look at the variable and then we say, "What is that variable?" If it's response one, then we run this set of commands. If it's a second response, we run this set of commands. If it's a third response, we run this set of commands. And notice that we end the case statement with case spelled backwards, 'esac'.

We actually did that with if-then-else; I failed to point that out. When you're done with the if-then-else statement in the script, you end it by spelling if backwards, 'fi'. You do the same thing with case; you spell case backwards, 'esac'. That tells the script, "Hey, we're done with the case statement."

Let's look at a simple example.

A case Example 7:24-9:16

Here, we have a script that asks users what county they were born in in a particular state. Based upon the response they give, we can cause the script to provide a customized response using a case statement.

The first thing this script does is ask the user what county they were born into, and it reads their response into a variable called mycoun. Then, using the value of mycoun, the case statement determines what part of the state they were born in using a case structure. We reference the mycoun variable in the case statement, and then we provide the different responses.

Notice, here, that we have three separate responses. They are separated by pipe characters; that means "or." If the value of mycoun equals Bingham, Bonneville, or Bannock, then we run the echo command, saying, "Being born in whatever county they entered, you were born in eastern Idaho. If the value of the mycoun variable is Ada, Owyhee, or Canyon, we run the echo command, saying, "Being born in whatever county you entered, you were born in western Idaho." We do the same thing for Cache, Oneida, and Franklin, tell them that they were born in southern Idaho. And we do the same thing, again, for Bonner, Shoshone, and Kootenai, which says that you were born in northern Idaho.

Notice, down here, that we have this response, this star (*). This is very important because within the case statement, it is possible that they either provided a response that is not included in any of the responses here, or that they, maybe, misspelled their response, in which case, the star, here, says to basically anything else--it's a catch-all here--"I'm sorry, I'm not familiar with that county."

The way the case statement works is it starts working through the list of responses. As soon as it hits a match, it ignores everything else and just runs whatever commands it's supposed to run. If it makes it all the way through the list-- meaning that no matches were made on any of these-- it will match on this, because the star sign says, "Match anything! I don't care what it is!" And then we use the echo command to put this text on the screen.

The while Loop 9:15-9:54

If-then-else, as well as case structures, are called branching structures because, depending upon how a condition evaluates, the script branches in one direction or another. You can also use a different kind of structure in your scripts called a looping control structure. Looping control structures come in three different varieties. We have the while loop, the until loop, and the for loop.

Let's take a look at the while loop first. A while loop executes over, and over, and over until the specified condition here is no longer true. So we say, while whatever condition it is that we specify here is true, run these commands. When that condition becomes false, then we exit out of the loop.

The until Loop 9:55-10:27

In addition to a while loop, you can also use an until loop in your script, and it works in the opposite manner of a while loop. An until loop uses a condition as well, just like a while loop does. But it works in the opposite manner. An until loop will run over, and over, and over, as long as this condition, here, is false. Remember, with a while loop, it runs as long as that condition is true--just the opposite with an until loop. As long as that condition is false, these commands will run. When that condition becomes true, we exit out of the loop and continue on.

The for Loop 10:28-11:07

In addition to a while loop and an until loop, you can also use a for loop. A for loop operates in a very different manner than while or until. Remember, the until and while loops will just keep looping indefinitely until the specified condition is met. A for loop, on the other hand, will loop a specified number of times. The structure is 'for i in' whatever we specify, do run these commands. And it will keep looping a certain number of times. You'll notice, here, that we use the sequence command, seq. It's very common to use a sequence command within a for loop to create a sequence of numbers that determines how many times it's going to loop.

The seq Command 11:08-12:32

There are three different ways for creating a number sequence with the sequence command. If I just specify a single value--say I type 'seq 10'--then the sequence is going to start at one, and then it's going to increment by one and end at the value I specified, so it will go one, two, three, and so on, until it reaches 10, and then it will stop. This will cause the for loop to loop ten times. Then it's going to exit out of the loop.

You can also use the sequence command with two numbers. For example, if I enter 'seq 5 10' (with a space in between 5 and 10), then the sequence is going to start at the first value, and then increment by one, and then end at the second value. In this case, the first number generated by sequence would actually not be one, but five, and then we'd go six, seven, and so on until it reaches ten, and then it would end.

You can also specify three values with the sequence command, say, 'seq 1 2 10'. This tells the sequence command to start at the first value, 1, but increment by two, and end at 10. In this case, it would start at 1. But then, because we're incrementing by two, the next number would be 3, 5, and so on. It would actually end at 9 in this case, because it can't go to 10 because it increments. The next increment would be 11, which is beyond the ending range.

The for Loop and seq Command 12:33-13:23

In this example, we're going to create a sequence of numbers from 1 to 15 because we only specified one number. We're going to start at 1 and we're going to increment by one until we reach 15. Notice that the value that we generate with the sequence command is going to be assigned to this variable, right here, named i. Each time we go through the loop, i is going to increment by one.

The first time we run it, it's going to be 1. The second time we run it, i is going to be 2. The third time through the loop, i is going to be 3, and so on. And we actually reference that value down here, with the echo command. We say what the current number in the sequence is followed by whatever the current value of i is.

When we run this script, we would have 15 lines in the output, each one saying, "The current number in the sequence is," and it would start at 1; then, the next line, the value of i would be 2; the third line, the value of i would be 3; and so on, until it hits 15, in which case, the for loop

would break.

Endless Loop 13:24-13:51

The biggest danger with looping structures is that it is possible to actually get stuck in the loop, where it loops over, and over, and over, and over, and never breaks. We call this an infinite loop. This happens when the condition never changes to a value that will break the loop. In this situation, your script gets hung because it just keeps running that same looping structure over, and over, and over, and over, and over. It will continue to do so for eternity until you manually break out of it using the Ctrl+C key.

That's it for this lesson.

Summary 13:51-13:59

In this lesson, we looked at several control structures that you can use within a shell script. We first looked at the if-then-else structure. We looked at the case structure. Then we looked at looping structures: while, until, and for.
