14.2.2 Bash Shell Parameters, User Variables and Expansions

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

Bash Shell Parameters, User Variables, and Expansions 0:00-0:39

Like all programming languages, the bash shell uses variables. A variable is like a container in memory. This container has a name, and it stores data. The data is called a value, and it can be a number, a single character, a string of characters, or an array of values. When a bash script runs, it can access environment variables and shell variables.

In this lesson, we're going to discuss some building blocks used to write shell scripts. We'll cover two unique types of variables that you can use in shell scripts, positional parameters, and special parameters. Then we're going to talk about user variables and a shell construct that looks and functions like a variable, a shell expansion.

Positional Parameters 0:40-2:12

First, let's talk about positional parameters. Positional parameters are a series of special variables that contain the arguments given when the shell is invoked. The names of these special variables are digits, so they're referenced by $1, $2, $3, $4, and so on. Let's look at this example.

We have a shell script in an executable file named testparams. Notice that the first three positional parameters, $1, $2 and $3, are referenced in the script. When we run this script, $1 is set to the first argument, ‘TestOut'; $2 is set to the second argument, ‘is'; and $3 is set to the third argument, ‘Great'.

Your script can take different actions depending on the arguments you use when you run it. This is a much simpler than having your script prompt you for this information.

There's a couple of things you should be cautious about when you're using positional parameters. First, you can reset positional parameters within a script using the set command. If you have a long script, you might consider storing the position parameters in another variable near the start of a script. We'll describe how to do this in a minute.

Secondly, if you have more than nine arguments, the name of your positional parameter will use two digits, like $10, $11, etc. When you use $10, the bash shell interprets this as the $1 parameter followed by a zero (0). To get around this issue, you can enclose the parameter in braces like this: ${10}. This is a shell expansion, and I'll describe it in more detail later.

Special Parameters 2:13-3:06

Here is a list of special parameters that can only be referenced. Their values are set and maintained by the bash shell. Notice that when you reference $* and $@, they both contain the positional parameters. The difference is that $* is a single string with the positional parameters separated by a space, and $@ is an array with multiple values. Each positional parameter is added as a separate element of the array.

The hash (#) parameter also relates to positional parameters. When you reference $#, it expands to the number of positional parameters. The zero (0) parameter is like a positional parameter; it expands to the name of the shell when referenced in an interactive shell. In a script, $0 expands to the filename of the shell script.

We won't cover the rest of these special parameters right now. But you might find them useful as you write more complex scripts.

User Variables 3:07-4:48

In the bash shell, user variables are treated the same as environment variables and shell variables. But consider creating them with lowercase names. That way, there is less chance of your environment and shell variables being overwritten with new values. In most cases, you create user variables at the same time as you assign them a value.

Whether at a shell prompt or within a script, enter the name of the variable, the equal sign (=), and then the value you want to assign the variable. For example, you can enter ‘mytraining=TestOut'. If the value has a space in it, use single or double quotes around the value as in this command, ‘myname="Jane Doe"'.

Later, you reference the variable using the dollar sign ($). When you put a dollar sign followed by a variable name in a bash command, the value of the variable is used instead of the variable name. In bash, we describe this as expanding the variable. For example, if you enter the echo mytraining command, the output is simply "mytraining". But if you enter ‘echo $mytraining', the output is "TestOut". This function is often used when manipulating variable. For example, ‘fullname="$firstname $lastname"' or ‘savepositionalparameter=$1'.

There are a few bash commands, like the read command, that will create a variable and assign a value to it. For example, ‘read myname' will create the myname variable, read a line of text from the keyboard, and assign it to the myname variable. Another way to create a variable is using the declare command. It can be used every time you create a variable, but it's mostly used with shell arithmetic.

Finally, you can delete a user variable using the unset command.

Integer Variables 4:49-5:26

All bash variables are stored as character strings. But if you use the ‘declare -i' command, the variable is treated as an integer. An integer is a whole number, like the numbers you use to count. An integer can be positive, negative, or zero. The declare -I command allows the variable to be assigned a value using shell arithmetic. Shell arithmetic uses the plus character (+) for addition, the minus or dash character (-) for subtraction, the asterisk character (*) for multiplication, and the slash character (/) for division. These are the four basic arithmetic operators. Look on the bash man page for other operators that do more complex arithmetic.

Shell Arithmetic 5:27-6:56

To use shell arithmetic, enter the variable and the equal sign (=), and then enter an arithmetic equation. The equation can be made up of variables and operators. You can also use parenthesis to specify the order that the arithmetic operations are performed in.

There are a few things you need to remember about equations. First, you don't have to use the dollar sign when you want to use the value for a variable. For example, if you follow the x=4 command with the declare -i y=x+4 command, the value assigned to y will be 8. Just for comparison, if you left out -i, the value assigned to y would be the literal characters, x+4.

Another thing to remember is that you can't use spaces in the equation. The bash shell will try to evaluate each part that is delimited by spaces as a command or an argument to a command. Also, if a variable has character data instead of numerical data, the bash shell will still evaluate the equation by substituting a zero for character data. For example, the a=badchars command followed by the declare -i z=a+7 command will assign z the value of 7.

Finally, even though a variable has been declared using the -i option, you can still assign character data to it. The declare -i b=John command will be accepted, and the echo $b command will return the output "John". However, once the variable is declared using the -i option, the variable will always allow shell arithmetic.

Storing Arrays in Variables 6:57-9:13

You can assign a variable multiple values by declaring it as an array. There are two types of arrays, indexed and associative. The declare -a namearray=(Bob Sue Joe Jane) command creates an indexed array of four elements. The first element has an index number of zero (0), and the value is the characters, "Bob". By default, index numbers begin with zero (0) instead of (1). The fourth element has an index number of three (3) and the value "Jane".

You can add to the indexed array using commands like namearray[7]=George. Notice that the index numbers don't have to be consecutive. That's a funny quirk with the bash shell, but sometimes it's helpful. To reference an element in the array, add the index number to the end of the variable name within brackets. Then enclose the whole expression with braces. For example, ‘echo ${namearray[2]}' would output "Joe", the third element of the array. The braces are needed because the bash shell interprets $namearray as the first element and [2] as a literal string, so ‘echo $indarray[2]' would output "Bob[2]", which is not what we wanted.

The declare -iA agearray=([Bob]=21 [Sue]=19 [Joe]=22 [Jane]=25) command creates an associative array. Notice the uppercase A in the option instead of the lowercase a. We also added the -i option to allow shell arithmetic. Again, you reference an element of the array using the variable name, the bracketed element name, and braces. For example, ‘echo ${agearray[Sue]}' would output "19".

Array elements can be used in shell arithmetic. The command ‘agearray[Bob]=agearray[Sue]+agearray[Joe]' would assign the value of 41 (Sue's age plus Joe's age) to the Bob element. The ‘echo ${agearray[Bob]}' would then output "41".

Global vs. Local Variables 9:14-10:45

To complete our discussion on user variables, let's talk about the concept of local versus global variables. Consider what happens when you declare a user variable, say, myvar, in an interactive shell and then execute a script that declares a variable with the exact same name, myvar.

To complicate this situation, what happens if this script calls a second script that doesn't declares a myvar variable, but tries to reference it?

The key to answering this riddle is to realize that myvar is a local variable. It's local to the bash shell where it was declared. Recalling that a new bash shell is created when a script is run, we can see that since the interactive myvar variable is only a user variable; it's not transferred to the shell that the first script runs in. The first script creates its own myvar variable. When the first script runs the second script, a new shell is created, but neither the interactive myvar nor the myvar in the first script are transferred to the shell. The second script will not be able to find myvar and will return either an error or blank results when myvar is referenced.

This is the motivation for environment variables. They're considered global variables since they're global to the shell where they're created. So is each child shell associated with an environment variable. When a child shell spawns a new shell, that new shell will have access to the same environment variables. However, a copy of the environment variables is given to the child shell. The child can add environment variables that can be passed to its children, but the parent shell will not see the added environment variables.

Shell Expansions 10:46-11:12

Let's spend just a few minutes on shell expansions. This topic can quickly become complex. We just want to introduce two simple expansions, variable expansion (known as parameter expansion) and command expansion (known as command substitution).

The dollar sign character introduces expansion. When you use it in a command, what follows the dollar sign is expanded. Parameter expansion uses the dollar-sign-braces construct (${}). Command substitution uses the dollar-sign-parentheses construct [$()].

Parameter Expansion 11:13-12:32

We already saw one case where parameter expansion was required to get the proper results when you reference an array element. We had to use the dollar sign and braces in the echo ${namearray[2]} command to display the value of an array element. But parameter expansion goes much further because you can add operators. Here's another example starting with the location="NY 10014" command. We add an offset operator in this command, echo ${location:3}. This is a sub-string expansion that starts at the offset that follows the colon, in this case, 3, and continues to the end of the value. We start counting our offset beginning with 0. So, beginning with an offset of 3 and continuing to the end, this gives us "10014."

And here's just one more parameter expansion using pattern substitution. The echo ${location/10014/10032} command looks for the pattern "10014" in the location value and substitutes "10032." So, the output from the command would be "NY 10032."

Command Substitution 12:33-14:30

Command substitution allows you to use the output of a command to replace the command itself. The expansion is initiated with the dollar sign and followed by a command enclosed in parentheses. The command within the parentheses is run, and the output is substituted. Often, command substitution is used as an argument for another command or to set a variable. Here are some simple examples.

Let's examine this command, ‘mv -t ./old $(find -maxdepth 1 -mtime +30)'. The results of the find -maxdepth -mtime +30 command is a list of files in the current directory that are over 30 days old. Suppose there are only two files that are older than 30 days, a.dat and b.dat. Command substitution would replace the $() construct with these two files. Essentially, the command becomes ‘mv -t ./old a.dat b.dat'.

You can use command substitution to set a variable. If you need to save last Friday's date, yo, can use the ‘lastfriday=$(date --date="last Friday")' command. You could then use this variable in your script. For example, ‘echo "The file was backed up on $lastfriday"'.

There's another construct that performs command substitution, but doesn't use the dollar sign. It's an older construct, but it's still used quite often. When a command is listed within backquotes (`), command substitution is performed. The backquote on a keyboard is usually to the left of the 1 key. Make sure you use 1, and not the single quote (‘) that is next to the Enter key. To illustrate, our variable example could have been written as ‘`date --date="last Friday"`'.

Summary 14:31-14:45

And that completes our lesson. In this lesson, we introduced position parameters and special parameters. We described user variables, including integer variables used in shell arithmetic and arrays that can contain multiple values. We ended by looking at two shell expansions, parameter expansions, and command substitutions.