Easy Shell Scripting

By Blessen Cherian

Introduction

Shell scripting can be defined as a group of commands executed in sequence. Let's start by describing the steps needed to write and execute a shell script:

Step 1: *Open the file using an editor (e.g., "vi" or "pico".)*

```
vi Firstshellscript.sh
```

Step 2: *All shell scripts should begin with "#!/bin/bash" or whatever other shell you prefer. This line is called the shebang, and although it looks like a comment, it's not: it notifies the shell of the interpreter to be used for the script. The provided path must be an absolute one (you can't just use "bash", for example), and the shebang must be located on the first line of the script without any preceding space.*

Step 3: *Write the code that you want to develop. Our first shell script will be the usual "Hello World" routine, which we'll place in a file called 'Firstshellscript.sh'.*

```
#!/bin/sh
echo "Hello World"
```

Step 4:*The next step is to make the script executable by using the "chmod" command.*

```
chmod 744 Firstshellscript.sh
```

or

```
chmod +x Firstshellscript.sh
```

Step 5: *Execute the script. This can be done by entering the name of the script on the command line, preceded by its path. If it's in the current directory, this is very simple:*

```
bash$ ./Firstshellscript.sh
Hello World
```

If you want to see the execution step-by-step - which is very useful for troubleshooting - then execute it with the '-x' ('expand arguments') option:

```
sh -x Firstshellscript.sh
+ echo 'Hello World'
Hello World
```

Category: Shell scripting

Shell scripting can be defined as a group of commands executed in sequence.

To see the contents of a script, you can use the 'cat' command or simply open the script in any text editor:

```
bash$ cat Firstshellscript.sh
#!/bin/sh
echo Hello World
```

Comments in a Shell

In shell scripting, all lines beginning with # are comments.

```
# This is a comment line.
# This is another comment line.
```

You can also have comments that span multiple lines by using a colon and single quotes:

```
: 'This is a comment line.

Again, this is a comment line.

My God, this is yet another comment line.'
```

Note: This will not work if there is a single quote mark within the quoted contents.

Variables

As you may or may not know, variables are the most significant part of any programming language, be it Perl, C, or shell scripting. In the shell, variables are classified as either system variables or user-defined variables.

System Variables

System variables are defined and kept in the environment of the *parent shell* (the shell from which your script is launched.) They are also called environment variables. These variable names consist of capital letters, and can be seen by executing the 'set' command. Examples of system variables are PWD, HOME, USER, etc. The values of these system variables can be displayed individually by "echo"ing the system variables. E.g., `echo $HOME` will display the value stored in the system variable HOME.

When setting a system variable, be sure to use the "export" command to make it available to the *child shells* (any shells that are spawned from the current one, including scripts):

```
bash$ SCRIPT_PATH=/home/blessen/shellscript
bash$ export SCRIPT_PATH
```

Modern shells also allow doing all this in one pass:

```
bash$ export SCRIPT_PATH=/home/blessen/shellscript
```

User-Defined Variables

These are the variables that are normally used in scripting - ones that you don't want or need to make available to other programs. Their names cannot start with numbers, and are written using lower case letters and underscores by convention - e.g. 'define_tempval'.

When we assign a value to a variable, we write the variable name followed by '=' which is immediately followed by the value, e.g., `define_tempval=blessen` (note that there must not be any spaces around the equals sign.) Now, to use or display the value in `define_tempval`, we have to use the `echo` command and precede the variable name with a '$' sign, i.e.:

```
bash$ echo $define_tempval
blessen
```

The following script sets a variable named "username" and displays its content when executed.

```
#!/bin/sh

username=blessen
echo "The username is $username"
```

Commandline Arguments

These are variables that contain the arguments to a script when it is run. These variables are accessed using $1, $2, ... $n, where $1 is the first command-line argument, $2 the second, etc. Arguments are delimited by spaces. $0 is the name of the script. The variable $# will display the number of command-line arguments supplied; this number is limited to 9 arguments in the older shells, and is practically unlimited in the modern ones.

Consider a script that will take two command-line arguments and display them. We'll call it 'commandline.sh':

```
#!/bin/sh

echo "The first variable is $1"
echo "The second variable is $2"
```

When I execute 'commandline.sh' with command-line arguments like "blessen" and "lijoe", the output looks like this:

```
bash$ ./commandline.sh blessen lijoe
The first variable is blessen
The second variable is lijoe
```

Exit status variable

This variable tells us if the last command executed was successful or not. It is represented by $?. A value of 0 means that the command was successful. Any other number means that the command was unsuccessful (although a few programs such as 'mail' use a non-zero return to indicate status rather than failure.) Thus, it is very useful in scripting.

To test this, create a file named "test", by running `touch test` . Then, "display" the content of the file:

```
bash$ cat test
```

Then, check the value of $?.

```
bash$ echo $?
0
```

The value is zero because the command was successful. Now try running 'cat' on a file that isn't there:

```
bash$ cat xyz1
bash$ echo $?
1
```

The value 1 shows that the above command was unsuccessful.

Scope of a Variable

I am sure most programmers have learned (and probably worked with) variables and the concept of `scope` (that is, a definition of where a variable has meaning.) In shell programming, we also use the scope of a variable for various programming tasks - although this is very rarely necessary, it can be a useful tool. In the shell, there are two types of scope: global and local. Local variables are defined by using a "local" tag preceding the variable name when it is defined; all other variables, except for those associated with function arguments, are global, and thus accessible from anywhere within the script. The script below demonstrates the differing scopes of a local variable and a global one:

```
#!/bin/sh

display()
{
    local local_var=100
    global_var=blessen
    echo "local variable is $local_var"
    echo "global variable is $global_var"
}

echo "======================"
display
echo "=======outside ========"
echo "local variable outside function is $local_var"
echo "global variable outside function is $global_var"
```

Running the above produces the following output:

```
======================
local variable is 100
global variable is blessen
=======outside ========
local variable outside function is
global variable outside function is blessen
```

Note the absence of any value for the local variable outside the function.

Input and Output in Shell Scripting

For accepting input from the keyboard, we use `read`. This command will read values typed from the keyboard, and assign each to the variable specified for it.

```
read <variable_name>
```

For output, we use the `echo` command.

```
echo "statement to be displayed"
```

Arithmetic Operations in Shell Scripting

Like other scripting languages, shell scripting also allows us to use arithmetic operations such as addition, subtraction, multiplication, and division. To use these, one uses a function called `expr`; e.g., "expr a + b" means 'add a and b'.

e.g.:

```
sum=`expr 12 + 20`
```

Similar syntax can be used for subtraction, division, and multiplication. There is another way to handle arithmetic operations; enclose the variables and the equation inside a square-bracket expression starting with a "$" sign. The syntax is

```
$[expression operation statement]
```

e.g.:

```
echo $[12 + 10]
```

[Note that this syntax is not universal; e.g., it will fail in the Korn shell. The '$((...))' syntax is more shell-agnostic; better yet, on the general principle of "let the shell do what it does best and leave the rest to the standard toolkit", use a calculator program such as 'bc' or 'dc' and command substitution. Also, note that shell arithmetic is integer-only, while the above two methods have no such problem. -- Ben]

Conditional Statements

Let's have some fun with a conditional statement like "if condition". Most of the time, we shell programmers have situations where we have to compare two variables, and then execute certain statements depending on the truth or falsity of the condition. So, in such cases, we have to use an "if" statement. The syntax is show below:

```
if [ conditional statement ]
then
        ... Any commands/statements ...
fi
```

The script cited below will prompt for a username, and if the user name is "blessen", will display a message showing that I have successfully logged in. Otherwise it will display the message "wrong username".

```
#!/bin/sh

echo "Enter your username:"
read username

if [ "$username" = "blessen" ]
then
        echo 'Success!!! You are now logged in.'
else
        echo 'Sorry, wrong username.'
fi
```

Remember to always enclose the variable being tested in double quotes; not doing so will cause your script to fail due to incorrect syntax when the variable is empty. Also, the square brackets (which are an alias for the 'test' command) must have a space following the opening bracket and preceding the closing one.

Variable Comparison

In shell scripting we can perform variable comparison. If the values of variables to be compared are numerical, then you have to use these options:

-eq	Equal to
-ne	Not Equal to
-lt	Less than
-le	Less than or equal to
-gt	Greater than
-ge	Greater then or equal to

If they are strings, then you have to use these options:

=	Equal to
!=	Not Equal to
<	First string sorts before second
>	First string sorts after second

Loops

The "for" Loop

The most commonly used loop is the "for" loop. In shell scripting, there are two types: one that is similar to C's "for" loop, and an iterator (list processing) loop.

Syntax for the first type of "for" loop (again, this type is only available in modern shells):

```
for ((initialization; condition; increment/decrement))
do
        ...statements...
done
```

Example:

```
#!/bin/sh

for (( i=1; $i <= 10; i++ ))
do
        echo $i
done
```

This will produce a list of numbers from 1 to 10. The syntax for the second, more widely-available, type of "for" loop is:

```
for <variable> in <list>
do
        ...statements...
done
```

This script will read the contents of '/etc/group' and display each line, one at a time:

```
#!/bin/sh

count=0
for i in `cat /etc/group`
do
         count=`expr "$count" + 1`
         echo "Line $count is being displayed"
         echo $i
done

echo "End of file"
```

Another example of the "for" loop uses "seq" to generate a sequence:

```
#!/bin/sh

for i in `seq 1 5`
do
         echo $i
done
```

While Loop

The "while" loop is another useful loop used in all programming languages; it will continue to execute until the condition specified becomes false.

```
while [ condition ]
do
         ...statement...
done
```

The following script assigns the value "1" to the variable `num` and adds one to the value of `num` each time it goes around the loop, as long as the value of `num` is less than 5.

```
#!/bin/sh

num=1

while [$num -lt 5]; do num=$[$num + 1]; echo $num; done
```

Category: Programming

[Break] code into small chunks called functions, and call them by name in the main program. This approach helps in debug

Select and Case Statement

Similar to the "switch/case" construct in C programming, the combination of "select" and "case" provides shell programmers with the same features. The "select" statement is not part of the "case" statement, but I've put the two of them together to illustrate how both can be used in programming.

Syntax of select:

```
select <variable> in <list>
do
        ...statements...
done
```

Syntax of case:

```
case $<variable> in
        <option1>) statements ;;
        <option2>) statements ;;
        *) echo "Sorry, wrong option" ;;
esac
```

The example below will explain the usage of select and case together, and display options involving a machine's services needing to be restarted. When the user selects a particular option, the script starts the corresponding service.

```
#!/bin/bash

echo "***********************"
select opt in apache named sendmail
        do
        case $opt in
                apache)  /etc/rc.d/init.d/httpd restart;;
                named)          /etc/rc.d/init.d/named restart;;
                sendmail)       /etc/rc.d/init.d/sendmail restart;;
                *)                      echo "Nothing will be restarted"
        esac
        echo "***********************"

        # If this break is not here, then we won't get a shell prompt.
        break

done
```

[Rather than using an explicit 'break' statement - which is not useful if you want to execute more than one of the presented options - it is much better to include 'Quit' as the last option in the select list, along with a matching case statement. -- Ben]

Functions

In the modern world where all programmers use the OOP model for programming, even we shell programmers aren't far behind. We too can break our code into small chunks called functions, and call them by name in the main program. This approach helps in debugging, code re-usability, etc.

Syntax for "function" is:

```
<name of function> ()
{       # start of function
        statements
}       # end of function
```

Functions are invoked by citing their names in the main program, optionally followed by arguments. For example:

```
#!/bin/sh
```

```
sumcalc ()
{
         sum=$[$1 + $2]
}

echo "Enter the first number:"
read num1
echo "Enter the second number:"
read num2

sumcalc $num1 $num2

echo "Output from function sumcalc: $sum"
```

Debugging Shell Scripts

Now and then, we need to debug our programs. To do so, we use the '-x' and '-v' options of the shell. The '-v' option produces verbose output. The '-x' option will expand each simple command, "for" command, "case" command, "select" command, or arithmetic "for" command, displaying the expanded value of PS4, followed by the command and its expanded arguments or associated word list. Try them in that order - they can be very helpful when you can't figure out the location of a problem in your script