Flow Control - Part 1

In this lesson, we will look at how to add intelligence to our scripts. So far, our project script has only consisted of a sequence of commands that starts at the first line and continues line by line until it reaches the end. Most programs do much more than this. They make decisions and perform different actions depending on conditions.

The shell provides several commands that we can use to control the flow of execution in our program. In this lesson, we will look at the following:

- **`if`**
- **`test`**
- **`exit`**

if

The first command we will look at is **`if`**. The **`if`** command is fairly simple on the surface; it makes a decision based on the *exit status* of a command. The **`if`** command's syntax looks like this:

The if statement has the following syntax:

```
if commands; then
commands
[elif commands; then
commands...]
[else
commands]
fi
```

where *commands* is a list of commands. This is a little confusing at first glance. But before we can clear this up, we have to look at how the shell evaluates the success or failure of a command.

Exit Status

Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution. By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status. Here we see it in action:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
```

```
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

In this example, we execute the **ls** command twice. The first time, the command executes successfully. If we display the value of the parameter **$?**, we see that it is zero. We execute the ls command a second time, producing an error and examine the parameter **$?** again. This time it contains a 2, indicating that the command encountered an error. Some commands use different exit status values to provide diagnostics for errors, while many commands simply exit with a value of one when they fail. Man pages often include a section entitled “Exit Status,” describing what codes are used. However, a zero always indicates success.

The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The **true** command always executes successfully and the **false** command always executes unsuccessfully:

```
[me@linuxbox~]$ true
[me@linuxbox~]$ echo $?
0
[me@linuxbox~]$ false
[me@linuxbox~]$ echo $?
1
```

We can use these commands to see how the **if** statement works. What the **if**

statement really does is evaluate the success or failure of commands:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

The command `echo "It's true."` is executed when the command following **`if`** executes successfully, and is not executed when the command following **`if`** does not execute successfully.

test

The **`test`** command is used most often with the **`if`** command to perform true/false decisions. The command is unusual in that it has two different syntactic forms:

```
# First form

test expression

# Second form

[ expression ]
```

The **test** command works simply. If the given expression is true, **test** exits with a status of zero; otherwise it exits with a status of 1.

The neat feature of **test** is the variety of expressions you can create. Here is an example:

```
if [ -f .bash_profile ]; then
    echo "You have a .bash_profile. Things are fine."
else
    echo "Yikes! You have no .bash_profile!"
fi
```

In this example, we use the expression " **-f .bash_profile** ". This expression asks, "Is .bash_profile a file?" If the expression is true, then **test** exits with a zero (indicating true) and the **if** command executes the command(s) following the word **then**. If the expression is false, then **test** exits with a status of one and the **if** command executes the command(s) following the word **else**.

Here is a partial list of the conditions that **test** can evaluate. Since **test** is a shell builtin, use "**help test**" to see a complete list.

Expression	Description
-d *file*	True if *file* is a directory.
-e *file*	True if *file* exists.
-f *file*	True if *file* exists and is a regular file.
-L *file*	True if *file* is a symbolic link.
-r *file*	True if *file* is a file readable by you.
-w *file*	True if *file* is a file writable by you.

-x *file*	True if *file* is a file executable by you.
file1 -nt *file2*	True if *file1* is newer than (according to modification time) *file2*
file1 -ot *file2*	True if *file1* is older than *file2*
-z *string*	True if *string* is empty.
-n *string*	True if *string* is not empty.
string1 = *string2*	True if *string1* equals *string2.*
string1 != *string2*	True if *string1* does not equal *string2.*

Before we go on, I want to explain the rest of the example above, since it also

Before we go on, I want to explain the rest of the example above, since it also reveals more important ideas.

In the first line of the script, we see the **if** command followed by the **test** command, followed by a semicolon, and finally the word **then**. I chose to use the **[** ***expression*** **]** form of the **test** command since most people think it's easier to read. Notice that the spaces required between the "**[**" and the beginning of the expression are required. Likewise, the space between the end of the expression and the trailing "**]**".

The semicolon is a command separator. Using it allows you to put more than one command on a line. For example:

```
[me@linuxbox me]$ clear; ls
```

will clear the screen and execute the ls command.

I use the semicolon as I did to allow me to put the word **then** on the same line as the **if** command, because I think it is easier to read that way.

On the second line, there is our old friend **echo**. The only thing of note on this line is the indentation. Again for the benefit of readability, it is traditional to indent all blocks of conditional code; that is, any code that will only be executed if certain conditions are met. The shell does not require this; it is done to make the code easier to read.

In other words, we could write the following and get the same results:

```
# Alternate form

if [ -f .bash_profile ]
then
    echo "You have a .bash_profile. Things are fine."
else
    echo "Yikes! You have no .bash_profile!"
fi

# Another alternate form

if [ -f .bash_profile ]
then echo "You have a .bash_profile. Things are fine."
else echo "Yikes! You have no .bash_profile!"
fi
```

exit

In order to be good script writers, we must set the exit status when our scripts finish. To do this, use the **exit** command. The **exit** command causes the script to terminate immediately and set the exit status to whatever value is given as an

argument. For example:

```
exit 0
```

exits your script and sets the exit status to 0 (success), whereas

```
exit 1
```

exits your script and sets the exit status to 1 (failure).

Testing For Root

When we last left our script, we required that it be run with superuser privileges. This is because the **home_space** function needs to examine the size of each user's home directory, and only the superuser is allowed to do that.

But what happens if a regular user runs our script? It produces a lot of ugly error messages. What if we could put something in the script to stop it if a regular user

attempts to run it?

The **id** command can tell us who the current user is. When executed with the "-u" option, it prints the numeric user id of the current user.

```
[me@linuxbox me]$ id -u
501
[me@linuxbox me]$ su
Password:
[root@linuxbox me]# id -u
0
```

If the superuser executes **id -u**, the command will output "0." This fact can be the basis of our test:

```
if [ $(id -u) = "0" ]; then
    echo "superuser"
fi
```

In this example, if the output of the command **id -u** is equal to the string "0", then

print the string "superuser."

While this code will detect if the user is the superuser, it does not really solve the problem yet. We want to stop the script if the user is not the superuser, so we will code it like so:

```
if [ $(id -u) != "0" ]; then
    echo "You must be the superuser to run this script" >&2
    exit 1
fi
```

With this code, if the output of the **id -u** command is not equal to "0", then the script prints a descriptive error message, exits, and sets the exit status to 1, indicating to the operating system that the script executed unsuccessfully.

Notice the ">&2" at the end of the **echo** command. This is another form of I/O direction. You will often notice this in routines that display error messages. If this redirection were not done, the error message would go to standard output. With this redirection, the message is sent to standard error. Since we are executing our script and redirecting its standard output to a file, we want the error messages separated from the normal output.

We could put this routine near the beginning of our script so it has a chance to detect a possible error before things get under way, but in order to run this script as

an ordinary user, we will use the same idea and modify the `home_space` function to test for proper privileges instead, like so:

```
function home_space
{
    # Only the superuser can get this information

    if [ "$(id -u)" = "0" ]; then
        echo "<h2>Home directory space by user</h2>"
        echo "<pre>"
        echo "Bytes Directory"
            du -s /home/* | sort -nr
        echo "</pre>"
    fi

}   # end of home_space
```

This way, if an ordinary user runs the script, the troublesome code will be passed over, rather than executed and the problem will be solved.
