# 14.1.2 Bash Script Execution

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

Script Execution 0:00-0:13

In this lesson, we're going to look at how you execute shell scripts from the shell prompt. This is very important because if you don't configure your shell scripts to be executable, then they won't run.

---

Options for Running a Shell Script 0:14-1:28

For example, if I had a script file that I created, and I named it showdate, and I saved it in my /home directory and then tried to run it from the shell prompt, absolutely nothing would happen--except for an error message on the screen. This happens because by default, the Bash shell sees this file as a text file. You know what? You can't run a text file.

There are two options for getting around this. The first one is to actually call the shell executable itself, which is /bin/bash, and then tell it to execute the script file. In this example, at the shell prompt, I would enter '/bin/bash' and then tell it to tell the ‘showdate' script located in my user's /home directory. This works, but it really isn't the best option. If you're going to be running your scripts, yeah, this would probably work okay. But if you expect some other user to be able to run your scripts, then, more than likely, they're not going to know they need to call the shell executable itself and run the script within it.

A better option is to go ahead and assign the execute permission to the text file. With the execute permission assigned, any file--including a simple text file--can be allowed to execute from the shell prompt, and this is a great option for making scripts easy for end users to run.

---

Assign the Execute Permission 1:29-3:30

To do this, you simply enable the execute permission for owner, for group, for others, or any combination thereof. This is done using the chmod command. In the example we're working with here, I can allow this showdate file to be executed by the Bash shell by adding the execute permission to whatever users we want to be able to run it.

For example, let's suppose that I only care about allowing myself, the script file owner, to execute it. To do this, all I have to do is add the execute permission to the owning user. I could use this command to accomplish this. I enter 'chmod u', which stands for user, or the file owner, '+x'. That tells chmod to add the execute permission to the owner. An example of that is shown down here. I've entered 'chmod u+x showdate'. And notice, down here, in the output of the ls -l command, that the file owner now has the execute permission. You could accomplish this same thing using numeric permissions as well. Just use 'chmod 744 showdate', and that will also add the execute permission to the file owner.

Remember, when we calculate permissions numerically, read has a value of 4, write has a value of 2, and execute has a value of 1. So, if I want my user--the file owner--to have read, write, and execute permissions, I just add these together: 4 + 2 + 1, = 7. Because group and others have 4, they're allowed to look at the file. They have read access. But they're not allowed to write to it, nor are they able to execute it.

With the execute permission added to owner, I can then run this script from the shell prompt. All I have to do is enter its name, just as I would any other system command. Of course, you could allow group or others to execute the file as well. Just do a careful evaluation of who you want to allow to execute the script and assign the appropriate permissions.

---

Enter the Full Path 3:31-4:59

There is one other issue that you have to be aware of when you're working with scripts. That is the issue of paths. Notice, in this example, that in order to run the showdate command, which resides in my /home directory, I had to enter basically the full path to that command. I entered a './', which indicates the current directory, and then the name of the file.

Even though the current directory is my /home directory, I have changed to the directory where the showdate command resides. Even though that is the case, if I do not provide the full path to the showdate file--as in this example--even though I'm in the same directory where showdate exists, if I try to run it from the shell prompt, I get a command not found error.

Here's the problem. My /home directory (which, in this case, is /home/rtracy), this directory is not in my PATH environment variable. If I had not explicitly indicated the path to the script file in the first example here, then I see the error. If I do supply the full path to the file, then the Bash shell knows where it's at, and it doesn't have to look through the PATH environment variable trying to find it.

Here's the key thing to remember: if the script resides in a directory that is not found in the PATH environment variable, you must specify the full path to that script file in the shell command; otherwise, the Bash shell has no idea where that file actually exists.

---

## Directory Within PATH 5:00-6:08

If you're creating scripts for yourself, scripts that you will just run, this probably doesn't pose a problem. If you're creating a script that you want your end users to be able to run, then this is a problem. I can guarantee you that you'll get complaints because they won't remember to include the full path to the script file. To save yourself frustration, I recommend that you consider putting the script file in a directory that is part of your PATH environment variable.

For example, one option would be to place the script in the /bin subdirectory within each user's /home directory. Most Linux distributions automatically create a directory named bin, by default, within each user's /home directory. Then one of the Bash configuration files automatically adds that directory to the PATH environment variable when the shell is originally started. You can see that /home/rtracy/bin is in the path. Then, because it's in the path, any files that I dump into this directory can be run from the shell prompt without specifying the full path to the file.

---

## Add Directory to PATH 6:09-7:13

Alternatively, you could do just the opposite. You could take the directory where that script file resides and add it to the existing PATH environment variable.

To do this, you need to use two different commands. First, we use PATH= to set the value of the PATH environment variable. Notice, here, that we first grab whatever the current value of the PATH environment variable is. Otherwise, we will overwrite the PATH environment variable, and nothing will work properly on this system after doing so. We, first of all, reference the existing PATH environment variable with '$PATH'. Then we append to the end a colon (:) followed by the new directory path that we want to add, and then we export that path.

When you do this, it is absolutely critical that you don't forget to include the existing path assignment in the new value of the PATH environment variable. If you don't do this, then the current directories in your PATH environment variable will be erased and will be replaced by whatever path you specify in the command. When this happens, your system is going to start to experience a host of problems, and you're not going to be happy.

---

## Export PATH 7:14-7:57

At this point, we've added the path to our environment variable, but we still have one other thing we need to do. We've assigned the value of PATH to include the additional directory, but that new value of PATH will only apply to the current shell session. That means if I were to open up another terminal session, I would see that the changes I made to the PATH environment variable, right here, would not be applied.

In order to make this assignment apply to all shell sessions, I need to export the new PATH environment variable. To do this, you just enter 'export PATH' at the shell prompt. After I do this, then the new value that I assigned to the PATH environment variable will be made available to all other shell sessions, including any sub-shells that are created by the current shell.

---

## Persist PATH Across Restarts 7:58-9:44

One problem that you're going to encounter in this process is the fact that any new value you add to the PATH environment variable will be lost after the system reboots. More than likely, you're not going to want that to happen. More than likely, you're going to want this change to be persistent across system restarts. In order to do this, we need to edit one of our Bash configuration files and add these two variable assignment commands to that file to make sure that every time the system starts, that change is applied.

At this point, you have to decide which shell configuration file is the right one to use. It depends upon two things. Number one, it depends upon your distribution because some distributions use one set of files; other distributions use a completely different set of shell configuration files. You have to look at your distribution to figure out which one it uses, and then you have to decide the scope of this change. Do you want it to apply to just one user on the system, maybe your user account? Or do you want this change to be applied to all users on the system?

If you want this change to apply to all users, then you need to make the change to a global Bash configuration file, such as the /etc/profile file. This will cause this change to be applied to all users, regardless of who it is that logs in.

If, on the other hand, you only care about making this change for a single user, such as yourself, then you just find the appropriate hidden configuration file within your user's /home directory, such as .bashrc, and make that change there. Then it will only be applied to your particular user account. Everyone else will not receive the change.

---

Summary 9:45-9:49

That's it for this lesson. In this lesson, we talked about how to execute a shell script, and we also talked about how to deal with PATH issues when executing a shell script.

---