Scripts and Startup

This lesson will cover two related subjects: Shell scripts and what happens when the shell is started up.

WE'LL COVER THE FOLLOWING

You've probably come across shell scripts, which won't take long to cover, but shell **startup** is a useful but tricky topic that catches most people out at some point, and is worth understanding well.

How Important is this Lesson?

This lesson is essential to the course.

Shell Scripts

First create a folder to work in and move into it:

Terminal

A shell script is simply a collection of shell commands that can be run non-interactively. These can be quick one-off scripts that you run, or very complex programs.

The Shebang

Run this:

```
echo '#!/bin/bash' > simple_script
echo 'echo I am a script' >> simple_script
```

Type the above code into the terminal in this lesson.

You have just created a file called `simple_script` that has two lines in it. The first consists of two special characters: the hash and the exclamation mark. This is often called *shebang*, or *hashbang* to make it easier to say. When the operating system is given a file to run as a program, if it sees those two characters at the start, it knows that the file is to be run under the control of another program (or *interpreter* as shells are often called).

Now try running it:

```
./simple_script
```

Type the above code into the terminal in this lesson.

That should have failed. Before we explain why, let's understand the command.

The `./` characters at the start of the above command tells the shell that you want to run this file from within the context of the current working directory. It's followed by the filename to run.

Similarly, the `../` characters indicate that you want to run from the directory above the current working directory.

This:

```
mkdir tmp
```

```
cd tmp
../simple_script   # Call the script from the tmp subfolder


cd ..
rm -rf tmp
```

Type the above code into the terminal in this lesson.

will give you the same output as before.

The Executable Flag

The script in the above example will have failed because the file was not marked as *executable*, so you will have got an error saying permission was denied.

To correct this, run:

```
chmod +x simple_script
./simple_script
```

Type the above code into the terminal in this lesson.

The `chmod` program changes the permissions (or *modes* on a file), so that only certain users, or groups of users can *read*, *write*, or *execute* (ie run as a program) a file.

> Note: The subject of file permissions and ownership can get complex and is not covered in full here. `man chmod` is a good place to start if you are interested.

The `PATH` Variable

What happens if you don't specify the `./` and just run:

```
simple_script
```

Type the above code into the terminal in this lesson.

Either you'll get an error saying the file cannot be found, or the command might work as before.

The reason the expected output is known is because it depends on how the `PATH` variable is set up in the environment.

If you run the code below you will see your `PATH` variable:

```
echo $PATH
```

Type the above code into the terminal in this lesson.

Your output may vary. Here is an example:

```
/home/imiell/perl5/bin:/opt/local/bin:/opt/local/sbin:/usr/local/bin:/usr/
bin:/bin:/usr/sbin:/sbin:/opt/X11/bin
```

The `PATH` variable is a set of directories, separated by colons. It could be as simple as:

```
/usr/sbin:/usr/bin
```

for example.

These are the directories bash looks through to find commands, in order.

So what sets up the `PATH` variable if you did not? The answer is: bash startup scripts.

But before we discuss them, how can we make sure that simple_script can be run without using `./` at the front?

```
PATH=${PATH}:.
simple_script
```

Type the above code into the terminal in this lesson.

That's how! In the first line you set the `PATH` to itself, plus the current working directory. It then looks through all the directories that were previously set in your `PATH` variable, and then finally tries the `.`, or local folder, as we saw before.

Startup Scripts

Understanding startup scripts and environment variables are key to a lot of

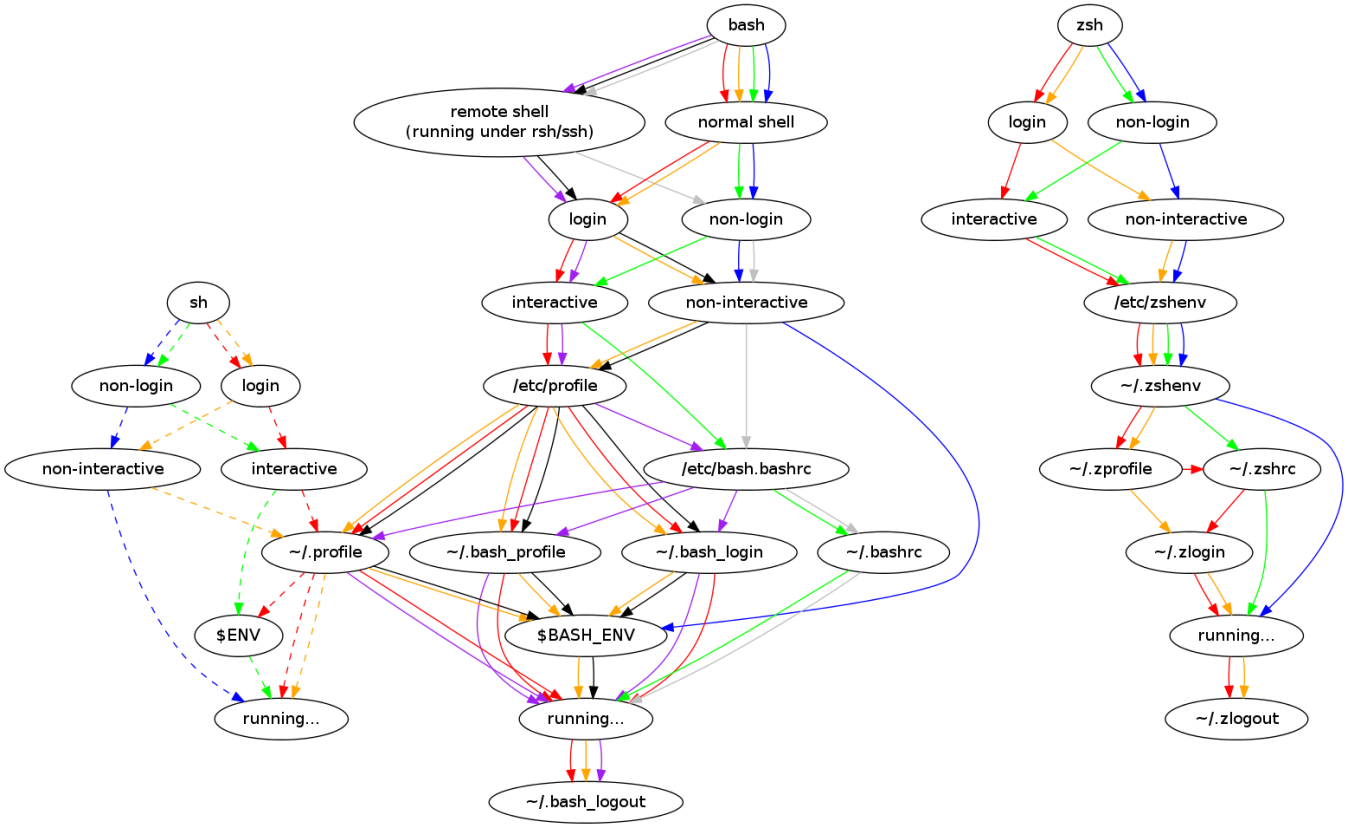
issues that you can end up spending a lot of time debugging! If something

works in one environment and not in another, the answer is often a difference in startup scripts and how they set up an environment.

Startup Explained

When bash starts up, it calls and runs a series of files to set up the environment you arrive at, at the terminal. If you've ever noticed that bash can 'pause' before giving you a prompt, it may be because the startup script is performing a command in the foreground.

Have a look at this diagram:



Yes, this can be confusing.

The diagram shows the startup script order for different shells in different contexts. Each context is shown by following a separate color through the diagram.

We are going to follow (from the top) the path from the 'bash' bubble, and ignore the 'zsh' and 'sh' paths, but it's interesting to note they have their own separate paths (in the case of 'zsh') and join up at points (in the case of 'sh' and 'bash').

At each point in this graph the shell you choose either makes a decision about which path to follow (eg whether the shell is *interactive* or not), or runs a script if the color has already been determined from these decisions.

> Note: interactive means running with a terminal attached to the process. If a script is run, and you can't type input to it, it is *non-interactive*.

We'll walk through this below, which should make things clearer.

When You Run Bash

So which path does it take when you run `bash` on the command line?

- The first decision you need to make is whether bash is running *locally* or as a *remote* shell on another server
- From there, you decide if this is a *login* or a *non-login* shell. You did not login when you ran bash, so follow 'non-login'
- The final decision is whether bash is running interactively (ie can you type things in, or is bash running a script?). You are on an interactive shell, so follow 'interactive'
- Now, whichever colour line you have followed up to this point, continue with it: those files are the ones that get run when bash is started up
- If the file does not exist, it is simply ignored

Beware

To *further* complicate things, these scripts can be made to call each other in ways that confuse you if you simply believe that diagram. So be careful!

The `source` Builtin

Now that you understand builtins, shell scripts, and environments, it's a good time to introduce another builtin: `source`.

```
MYVAR=Hello
echo 'echo $MYVAR' > simple_echo  # Write a program that outputs a variable
chmod +x simple_echo              # Make the script executable
./simple_echo                     # Call the script, which outputs nothing as MYVAR is not ex
source simple_echo                # source the script, which actually outputs the MYVAR vari
```

```
source simple_echo                # source the script, which actually outputs the MYVAR varia
```

Type the above code into the terminal in this lesson.

From the above example, you can see that the source runs the script from within the same shell context.

So if you put just `exit` within the code you run, then a `source`d script will exit the shell you are in, rather than the script that is being run!

If you didn't follow what the above script did when sourced, do it again, and think carefully about what each command is doing.

> Note: Most shell scripts have a `.sh` suffix, but this is not required - the OS does not care or take any notice of the suffix.

Avoiding Startup Scripts

It's often useful to start up bash while avoiding all these startup scripts, to get bash in as raw a state as possible. When debugging, or testing core bash functionality, this can be invaluable.

To acheive that, run this:

```
env -i bash --noprofile --norc
```

Type the above code into the terminal in this lesson.

`env` is a program that works on the environment. The effect of the `-i` flag is to remove the environment variables from the command you run. This means that exported variables will not get inherited by the bash program we are running. We're running the bash program itself with two flags. The `--noprofile` flag tells bash not to source the system-wide bash startup files, and the `--norc` tells bash not to source the personal ones present in your home folder.

The end effect of this is that your shell has a very minimal set of variables available:

```
env
```

Type the above code into the terminal in this lesson.

Type the above code into the terminal in this lesson.

Because you 'emptied' the environment, the number of variables you should see will be very small.

Scripts and Startup Quiz

1 Bash scripts must be suffixed with `.sh`. True or false?

COMPLETED 0%

1 of 2

What You Learned

- What the 'shebang' is
- How to create and run a shell script
- The significance of the `PATH` environment variable
- What happens when bash starts up
- What the builtin `source` does

What Next?

Next you look at **command substitution**, which allows you to easily capture the output of commands to variables for later use.

Exercises

1) Go through all the scripts that your bash session went through on startup in the terminal above. Read through them and try and understand what they're

doing. If you don't understand parts of them, try and figure out what's going on by reading `man bash`.

2) Go through the other files in that diagram that exist in the terminal above. Do as per 1).