Debugging

In this lesson, you will learn about bash flags useful for debugging, how to trace bash code, and useful resources for linting bash scripts.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

If you write, use or maintain bash of any complexity you'll want to know how to debug it!

Syntax Checking Options

Start by creating this simple (but broken) script:

```
cat > debug_script.sh << 'END'
#!/bin/bash
A=some value
echo "${A}
echo "${B}"
END
```

Type the above code into the terminal in this lesson.

Terminal

Now run it with the `-n` flag. This flag only parses the script, rather than running it. It's useful for detecting basic syntax errors.

```
bash -n debug_script.sh
```

Type the above code into the terminal in this lesson.

You can see it's broken. Fix it. Then run it:

```
bash debug_script.sh
```

Type the above code into the terminal in this lesson.

You'll see:

```
    [1]+  Done                    sleep 60
```

in the terminal.

Again, it reports the job number, this time with the status (`Done`), and the command that was originally run (`sleep 60`).

Controlling Jobs

Just like starting jobs, you can control jobs by sending signals to them.

Here you're going to start two jobs, one to sleep for two minutes, and the next for one second more (so we can distinguish between them).

```
bash -v debug_script.sh
```

Type the above code into the terminal in this lesson.

Try tracing to see more details about what's going on. Each statement gets a new line.

```
bash -x debug_script.sh
```

Type the above code into the terminal in this lesson.

Using these flags together can help debug scripts where there is an elementary error, or even just working out what's going on when a script runs. I used it only yesterday to figure out why a systemctl service wasn't running or logging.

Fix the error you see before continuing.

Managing Variables

Variables are a core part of most serious bash scripts (and even one-liners!), so managing them is another important way to reduce the possibility of your script breaking.

Change your script to add the 'set' line immediately after the first line and see what happens.

```
#!/bin/bash
set -o nounset
A="some value"
echo "${A}"
echo "${B}"
```

Now research what the `nounset` option does. Which `set` flag does this correspond to?

Show Hint

Without running this, try and figure out what this script will do. Will it run?

```
#!/bin/bash
set -o nounset
A="some value"
B=
echo "${A}"
echo "${B}"
```

Try it and see.

I always set `nounset` on my scripts as a habit. It can catch many problems before they become serious.

Profiling Bash Scripts

Returning to the `xtrace` (or `set -x`) flag, we can exploit its use of a PS variable to implement the profiling of a script:

```
#!/bin/bash
set -o nounset
set -o xtrace
declare A="some value"
PS4='$(date "+%s.%N => ")'
B=
echo "${A}"
A="another value"
echo "${A}"
echo "${B}"
ls
pwd
curl -q bbc.co.uk
true
```

Type the above code into the terminal in this lesson.

- **Lines 2-3** set the `nounset` and `xtrace` options.
- **Line 4** `declare`s a variable `A` and gives it a value.
- **Line 5** sets the `PS4` variable, which runs before each command is processed in bash, and outputs the result prepended by a `+` sign. From this point in the script, each line shows the time it was run to nanosecond granularity, allowing you to see where the time running the script goes.
- **Lines 6-14** runs various simple commands that exercise the ability to see how long each line runs for.

> Note: If you are on a Mac, then you might only get second-level granularity on the date!

Shellcheck

Finally, here is a very useful tip for understanding bash more deeply and improving any bash scripts you come across. Shellcheck is a website (http://www.shellcheck.net/) and a package that gives you advice to help fix

and improve your shell scripts. Very often, its advice has prompted me to research more deeply and understand bash better.

Here is some example output from a script I found on my laptop:

```
$ shellcheck shrinkpdf.sh
In shrinkpdf.sh line 44:
          -dColorImageResolution=$3             \
                                 ^-- SC2086: Double quote to prevent globb
ing and word splitting.

In shrinkpdf.sh line 46:
          -dGrayImageResolution=$3              \
                                ^-- SC2086: Double quote to prevent globbi
ng and word splitting.

In shrinkpdf.sh line 48:
          -dMonoImageResolution=$3              \
                                ^-- SC2086: Double quote to prevent globbi
ng and word splitting.

In shrinkpdf.sh line 57:
        if [ ! -f "$1" -o ! -f "$2" ]; then
                         ^-- SC2166: Prefer [ p ] || [ q ] as [ p -o q ] i
s not well defined.

In shrinkpdf.sh line 60:
        ISIZE="$(echo $(wc -c "$1") | cut -f1 -d\ )"
                       ^-- SC2046: Quote this to prevent word splitting.
                       ^-- SC2005: Useless echo? Instead of 'echo $(cm
d)', just use 'cmd'.

In shrinkpdf.sh line 61:
        OSIZE="$(echo $(wc -c "$2") | cut -f1 -d\ )"
                       ^-- SC2046: Quote this to prevent word splitting.
                       ^-- SC2005: Useless echo? Instead of 'echo $(cm
d)', just use 'cmd'.
```

The most common reminders are regarding potential quoting issues, but you can see other useful tips in the above output, such as preferred arguments to the `test` construct, and advice on 'useless' `echo`s.

What You Learned

In this lesson, you learned:

- bash flags useful for debugging

- How to use traps and `declare` to trace the use of variables
- How to make your scripts more robust with `nounset`
- How to use `shellcheck` to help you reduce the risk of your scripts failing

What Next?

Next you will look at *string manipulation* in bash.

Exercises

1) Find a large bash script on a social coding site such as GitHub, and run `shellcheck` over it. Contribute back any improvements you find.