The 'set' Builtin

In this lesson you will become familiar with the `set` builtin, which allows you to manipulate these options within your scripts. You will cover the 'set' builtin, what POSIX is, and some useful options to set when scripting.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

When using bash it is very important to understand what its **options** are, how to set them, and how this can affect the running of your scripts.

Setting options is not absolutely core material, but once you've used bash for a while, understanding them becomes important. It's also good revision material for some of the concepts covered previously regarding exported vs. non-exported variables.

Running `set`

Start by running `set` on its own:

```
set
```

Type the above code into the terminal in this lesson.

Terminal

This will produce a stream of output that represents the state of your shell. In the normal case, you will see all the variables and functions set in your environment.

But my bash man page says:

> Without options, the name and value of each shell variable are displayed in a format that can be reused as input for setting or resetting the currently-set variables. Read-only variables cannot be reset. In posix mode, only shell variables are listed.
>
> — bash man page

> Note: The **Portable Operating System Interface(POSIX)** is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

Can you work out from your `set` output whether you are in posix mode?

It is likely that you are not. If so, type:

```
bash
set -o posix
set
exit
```

Type the above code into the terminal in this lesson.

and you will observe that the output no longer has functions in it.

The `-o` in **line 2** switched on the posix option in your bash shell. The same command with `+o` will switch it off. I have trouble remembering which is 'on' and which is 'off' almost every time!

> Note: If you did not have functions, then either no functions were set, or you were in posix mode already!

The commands above put you in a fresh bash shell so that we would revert to the previous state.

To see how all your options are set type this:

```
set -o
```

Type the above code into the terminal in this lesson.

and you will see the current state of all your options.

What you see are all the options bash can set. One of the exercises below is to try to understand what they all mean, but in this section we're only going to focus on a couple that I use all the time.

`set` vs `env`

One thing that can confuse people is that the output of `set` is similar to the output of `env`, but different.

```
set
env
```

Type the above code into the terminal in this lesson.

The difference is that *exported* variables are shown by env, not all the variables set in the shell.

Useful Options for Scripting

Where set becomes really useful to understand is in scripting.

For example, I set these three up every time I start writing a shell script:

```
set -o errexit
set -o xtrace
set -o nounset
```

Type the above code into the terminal in this lesson.

Although you don't need to be in a script for them to work.

- The `errexit` option tells bash to exit the script if any command fails
- The `xtrace` option outputs each command as it is being run. This is really useful for seeing what command was actually run if (for example) you are using variables within your commands. It also helps you see the order in which commands are being run
- The `nounset` option gets bash to throw an error if a variable is not set when it is referred to

To see how this works in practice, first create a file:

```
echo '#!/bin/bash
set -o errexit
set -o xtrace
set -o nounset
pwd
cd $HOME
cd -
echo $DOESNOTEXIST
echo "should not get here"' > ascript.sh
```

Type the above code into the terminal in this lesson.

- The `ascript.sh` file you just created (by a redirect on **line 9**) declares itself as a bash script on **line 1**
- It then sets the three flags we mentioned above. Then it outputs the current working directory (`pwd` on **line 5**)
- **Lines 6 and** 7 move to the `$HOME` folder and back again
- **line 8** tries to `echo` a variable that does not exist

After that file is created, make it executable and run it:

```
chmod +x ascript.sh
./ascript.sh
```

Type the above code into the terminal in this lesson.

When the script is run, you should see the commands after the `xtrace` option is set outputted with a `+` sign in front. Interspersed with these lines are the output that would normally be seen on the terminal without the `xtrace`

option being set.

What you should *not* have seen is the last echo line that outputs `should not get here`. Because the `DOESNOTEXIST` variable is unset, the script will have finished with an error before that line is read in. This is because the `nounset` option is set on **line 4** of the `ascript.sh` script.

You should be able to explain to someone else what's going on at each line typed in, and what the output of the above means.

Flags With `set` Instead of Names

For each `set` option, you can use a flag instead. For example, this:

```
set -e
set -x
```

Type the above code into the terminal in this lesson.

is the same as:

```
set -o errexit
set -o xtrace
```

Type the above code into the terminal in this lesson.

I generally prefer the name form rather than flag, just because it's easier to read.

When I start writing a script, I usually start with the following:

```
#!/bin/bash

set -o errexit
set -o xtrace
set -o unset

[... remainder of script ...]
```

The `unset` throws an error if a variable is unset when referenced. (The special variables `$@` and `$*` are exempt.)

The `pipefail` Option, Exit Codes and Pipelines

The `pipefail` Option, Exit Codes and Pipelines

One option worth mentioning (as it is frequently referred to) is the **pipefail** option:

```
touch afile.txt
set -o pipefail
grep notthere afile.txt | xargs
echo $?
```

Type the above code into the terminal in this lesson.

As you know, the grep that doesn't find anything to match returns an exit code of `1` (false). This output is passed to `xargs`, which always returns a zero (true).

So the question here is: what exit code should be returned? True, because `xargs` always returns true? Or false, because the grep returned a non-zero exit code?

The answer depends on the `pipefail` setting.

```
set +o pipefail
grep notthere afile.txt | xargs
echo $?
```

Type the above code into the terminal in this lesson.

When switched on (remember, `-o` is on, `+o` is off - yes, I find it confusing too), the `pipefail` option ensure that the error code of the last command returns a non-zero status. Since `grep` returns non-zero even when there's no 'error' as such, you can get surprising behaviour when using pipes and exit codes.

By default, `pipefail` is off, so the second outcome is the default one.

`set` vs `shopt`

Although we don't cover it in depth in this section, it's worth mentioning that there are two ways to set bash options from within scripts or on the command line. You can use the `set` builtin command, or the `shopt` builtin command. They both manipulate the behaviour of the shell, and differ for historical reasons. The set options are inherited, or borrowed, from other shells'

options, while the `shopt` ones (mostly) originated in bash.

Just to demonstrate one option that you might find useful, the `globstar` option allows you to use two asterisks to match all files in the local directory and all subdirectories:

```
shopt -s globstar
ls **
```

Type the above code into the terminal in this lesson.

If you have a lot of files in your subfolders, then it might take a long time to return!

‘Set’ Quiz

1 `set +x` switches an option ‘on’. True or False?

COMPLETED 0%

1 of 3

What You Learned

- What the `set` builtin is
- How to set an *option*
- The difference between `-o` (on) and `+o` (off)
- Some of the most-used and useful *options*

What Next?

Next we cover **process substitution**, a tricky but useful concept to master for sophisticated bash scripting.

Exercises

1) Read the man page to see what all the options are. Don't worry if you don't understand it all yet, just get a feel for what's there.

2) Set up a shell with unique variables and functions in the lesson terminal and use `set` to create a script to recreate those items in a fresh bash shell.