Exit Codes

In this lesson, you will find out what an exit code is, how you can set one in a script and a function, some conventions around exit codes, and other 'special' parameters in bash.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

Most bash scripts won't be completely comprehensible or safely written unless **exit codes** are understood.

What Is An Exit Code?

After you run a command, function or builtin, a special variable is set that tells you what the result of that command was. If you're familiar with HTTP codes like 200 or 404, this is a similar concept to that.

To take a simple example, type this in:

```
ls
echo $?
doesnotexist
echo $?
```

Type the above code into the terminal in this lesson.

Terminal

When that special variable `$?` is set to `0` it means that the previous command completed successfully (**line 2**). When the special variable `$?` is set to non-zero (in this case `127` it means that it did not complete successfully.

Standard Exit Codes

There are guidelines for exit codes for those that want to follow standards. Be aware that not all programs follow these standards (`grep` is the most common example of a non-standard program, as you will learn)!

Some key ones are:

- `0` - 'command successfully run' (OK)
- `1` - Used when there is an error but no specific number reserved to indicate what it was (general error)
- `2` - Misuse of shell `builtin` command
- `126` - Permission problem or command is not executable
- `127` - No file found matching this command
- `128` - Invalid exit argument given (eg exit 1.76)
- `128+n` - Process killed with signal `n`, eg 130 = terminated with signal 2

> Note: Signals will be covered in part 4

Since codes `3`-`125` are not generally reserved, you might use them for your own purposes in your applications.

Exit Codes and `if` Statements

So far so simple, but unfortunately (and because they are useful) exit codes can be used for many different reasons, not just to tell you whether the command completed successfully or not. Just as with exit codes in HTTP, the application can use exit codes to indicate something went wrong, or it can return a '200 OK' and give you a message directly.

Try to predict the output of this:

```
echo 'grepme' > afile.txt
grep not_there afile.txt
echo $?
```

Type the above code into the terminal in this lesson.

Did you expect that?

- **Line 1** created a file called `afile.txt`
- On **line 2** `grep` finished successfully (there was no segmentation or memory fault, memory exhaustion, it was not killed etc) but no lines were matched, and it returned `1` as an exit code (**line 3**).

`grep` uses the exit code `0` to mean 'matched', and the exit code `1` to mean 'not matched'.

In one way this is great, because you can write if statements like this:

```
if grep grepme afile.txt
then
    echo 'matched!'
fi
```

Type the above code into the terminal in this lesson.

In the above code, the `if` statement on **line 1** returns a zero exit code (true) if the `grep` command matched any lines, and outputs the echo on **line 3**.

While that is a great feature of `grep`, it means that you cannot be sure about what an exit code might mean about a particular program's termination. I have to look up the meaning of `grep`'s exit code nearly every time, and if I use a program's exit code I usually make sure to do a few tests first to be sure I know what is going to happen!

Setting Your Own Exit Code

If you are writing a script, you can set the exit code of the function by using the `exit` builtin. You can simulate this simply by entering a new bash process and then exiting from it.

```
bash
exit 67    # Exit the just-entered bash shell with exit code 67
echo $?    # Retrieve the exit code
```

Type the above code into the terminal in this lesson.

Line 3 above should output the value of the exit code you quit the bash shell you set (in this case, `67`.

If you are writing a function, you can set the exit code of the function by using the `return` builtin.

Type this:

```
function trycmd {
    $1
    if [[ $? -eq 127 ]]
    then
        echo 'What are you doing?'
        return 1
    fi
}
trycmd ls
trycmd doesnotexit
```

Type the above code into the terminal in this lesson.

- **Lines 1-8** create a function called `trycmd` that runs only the command passed in as the first argument on **line 2**. The `$1` is replaced by bash with the command, which bash then runs.
- **Line 3** compares the exit code of the command just run to `127` (the error code for 'command not found') and returns with a `1` error code **line 6** and a message asking what the user is doing (**line** 7).
- The last two lines (**lines 9-10**) are example invocations of `trycmd`, one with `ls`, and one with `doesnotexist`.

Other Special Parameters

The variable `$?` is an example of a **special parameter**. I'm not sure why they are called special parameters and not special variables, but it is perhaps to do with the fact that they are considered alongside the normal parameters of functions and scripts (`$1`, `$2` etc) as automatically assigned variables within these contexts.

Two of the most important are used below. As an exercise, try and figure out what they are from context:

```
ps -ef | grep bash | grep $$
sleep 999 &                  # The & here runs the command in the background
echo $!
```

Type the above code into the terminal in this lesson.

> Note: If you want to know more about background jobs (triggered by the `&` above), then see here.

If you're still stuck, have a look at the bash man page by running `man bash`.

Exit Codes Quiz

1 `$?` is an example of a/an

COMPLETED 0%

1 of 2

What You Learned

- What an *exit code* is
- Some standard *exit codes* and their significance

- Not all applications use *exit codes* in the same way
- How tests and *exit codes* work together
- Some *special parameters*

What Next?

Next you will learn about bash **options**, and the `set` builtin.

Exercises

1) Look up under what circumstances git returns a non-zero exit code.

2) Look up all the 'special parameters' and see what they do. Play with them. Research under what circumstances you might want to use them.