Tests

In this lesson, you'll learn some of the pitfalls and rules of thumb for practical usage of tests. You'll learn about how bash tests can be written, binary, unary, and logical operators, and if statements.

WE'LL COVER THE FOLLOWING

How Important is this Lesson?

Tests are a fundamental part of bash scripting, whether it’s on the command line in one-liners, or in much larger scripts or chains of commands.

What Are Bash Tests?

A test in bash is not a test that your program works. It’s a way of writing an expression that can be true or false.

Tests in bash are constructs that allow you to implement *conditional expressions*. They use square brackets (ie `[` and `]`) to enclose what is being tested.

For example, the simplest tests might be:

```
[ 1 = 0 ]   # Test that should fail
echo $?     # Non-zero output means 'un-true'
[ 1 = 1 ]   # Test that should succeed
echo $?     # Zero output means 'true'
```

Type the above code into the terminal in this lesson.

Terminal

> Note: The `echo $?` command above is a little mystifying at this stage if you've not seen it before. We will cover it in more depth in a lesson later in this 'Scripting Bash' section of the course. For now, all you need to understand is this: the `$?` variable is a special variable that gives you a number telling you the result of the last-executed command. If it returned true, the number will (usually) be `0'. If it didn't, the number will (usually) *not* be '0'.

Things get more interesting if you try and compare values in your tests. Think about what this will output before typing it in:

```
A=1
[ $A = 1 ]
echo $?
[ $A == 1 ]   # Does a double equals sign make any difference?
echo $?
[ $A = 2 ]
echo $?
```

Type the above code into the terminal in this lesson.

A single equals sign works just the same as a double equals sign. Generally I prefer the double one so it does not get confused with variable assignment.

What is '`[`', Really?

It is worth noting that `[` is in fact a builtin, as well as (very often) a program.

```
which [      # Tells you where the '[' program is
type [       # Tells you that '[' is also a builtin
```

Type the above code into the terminal in this lesson.

and that `[` and `test` are synonymous:

```
which test
man test    # Hit q to get out of the manual page.
man [       # Takes you to the same page.
```

Type the above code into the terminal in this lesson.

> Note: `which` is a program (not a builtin!) that tells you where a program can be found on the system.

This is why a space is required after the `[`. The `[` is a *separate command* and spacing is how bash determines where one command ends and another begins.

Logical Operators

What do you expect the output of this to be?

```
( [ 1 = 1 ] || [ ! '0' = '0' ] ) && [ '2' = '2' ]
echo $?
```

Type the above code into the terminal in this lesson.

Similar to other languages, '!' means 'not', `||` means 'or', `&&` means 'and' and items within '()' are evaluated first.

Note that to combine the binary operators `||` and `&&` you need to have separate `[` and `]` pairs.

From this point forward in this course I will use logical operators (particularly `&&`) where it's appropriate to. This should get you used to the idiom.

If you want to do everything in *one* set of braces, you can run:

```
[  1 = 1 -o  ! '0' = '0'  -a  '2' = '2' ]
echo $?
```

Type the above code into the terminal in this lesson.

You can use `-o` as an ‘or’ operator within the square brackets, `-a` for ‘and’ and so on. But you can’t use `(` and `)` to group within them.

If you’re not confused yet, you might be soon! If you are, try and re-read the above until you get it.

The `[[` Operator

The `[[` operator is very similar to the ‘test’ operator with *two* square brackets instead of one:

```
[[ 1 = 1 ]]
echo $?
```

Type the above code into the terminal in this lesson.

This confused me a lot for some time! What is the difference between `[` and `[[` if they produce such similar output?

The differences between `[[` and `[` are relatively subtle. Type these lines to see examples:

```
unset DOESNOTEXIST
[ ${DOESNOTEXIST} = '' ]
echo $?
[[ ${DOESNOTEXIST} = '' ]]
echo $?
[ x${DOESNOTEXIST} = x ]
echo $?
```

Type the above code into the terminal in this lesson.

- **Line 1** above should error because the variable `DOESNOTEXIST` does not exist
- Bash processes that variable in **line 2**, and ends up running: `[ = '' ]`
- This makes no sense to bash, so it complains! It’s expecting something on the left hand side of the empty quotes
- **Line 3** shows that it errored by returning a non-zero exit code

- **Line 4** (which uses the double brackets `[[`) tolerates the fact that the variable does not exist, and treats it as the empty string. It therefore resolves to: `[ '' = '' ]`
- And does not error (**line 5**), returning `0`.
- **Line 6** acts as a workaround. By placing an `x` on both sides of the equation, the code ensures that *something* gets placed on the left hand side: `[ x = x ]`
- which returns 'true' (**line** 7 returns `0`).

You can frequently come across code like this:

```
[[ "x$DOESNOTEXIST" = "x" ]]
```

where users have put quotes on both sides *as well as* an `x` and put in double brackets. Only one of these protections is needed, but people get used to adding them to their bash scripts as a superstition. And it doesn't seem to do any harm.

Once again, you can see understanding how quotes work is critical to bash mastery!

Oh, and `[[` doesn't like the `-a` (and) and `-o` (or) operators.

So `[[` can handle some edge cases that are not handled the same way when using `[`. There are some other differences, but I won't cover them here.

> Note: If you want to understand more, go to http://serverfault.com/questions/52034/what-is-the-difference-between-double-and-single-square-brackets-in-bash

Confused?

You're not alone. In practice, I follow most style guides and always use `[[` until there is a good reason not to.

If I come across some tricky logic in code I need to understand, I just look it up there and then, usually in the bash man page.

Unary and Binary Operators

There are other shortcuts related to `test` (and its variants) that it's worth knowing about. These take a single argument:

```
echo $PWD
[ -z "$PWD" ]            # -z is a unary operator that takes one argument
echo $?                  # Returns false, as PWD has content
unset DOESNOTEXIST
[ -z "$DOESNOTEXIST" ]
echo $?                  # Returns true, as "$DOESNOTEXIST" is evaluated to empty
[ -z ]                   # No argument?
echo $?                  # Returns true...
```

If your `$PWD` environment variable is set (it usually is), then the -z will return `false`. This is because `-z` returns true only if the argument is an empty string. Interestingly, this test is OK with no argument! Just another confusing point about tests...

There are quite a few unary operators so I won't cover them all here. The ones I use most often are `-a` and `-d`:

```
touch tests_file   # Create tests file
mkdir tests_dir
[ -a tests_file ]  # -a returns true if file exists
echo $?
[ -d tests_file ]  # -d returns false if the directory does not exist
echo $?            # A normal file is not a directory
[ -a tests_dir ]   # A directory is a type of file, so returns true
echo $?
[ -d tests_dir ]   # -d returns true if directory exists
echo $?
```

Type the above code into the terminal in this lesson.

These are called 'unary operators' (because they take one argument).

There are many of these unary operators, but the differences between them are useful only in the rare cases when you need them. Generally I just use `-d`, `-a`, and `-z` and look up the others when I need something else.

We'll cover 'binary operators', which work on two arguments, while covering types in bash.

Types

Type-safety (if you're familiar with that concept from other languages) does not come up often in bash as an issue. But it is still significant. Try and work out what's going on here:

```
[ 10 < 2 ]       # Throws an error, as single bracket tests don't handle '<'
echo $?
[ '10' < '2' ]   # Quotes do not help
echo $?
[[ 10 < 2 ]]     # Double bracket tests work
echo $?          # But it treats the input as strings, not integers
[[ '10' < '2' ]] # Same result when quoted as a string
echo $?
```

Type the above code into the terminal in this lesson.

From this you should be able to work out that the `<` operator expects strings, and that this is another way `[[` protects you from the dangers of using `[`.

If you can't work it out, then re-run the above and play with it until it makes sense to you!

Then run this:

```
[ 10 -lt 2 ]      # The -lt operator works in single brackets
echo $?
[ '10' -lt '2' ]
echo $?           # Still treats arguments as integers when quoted
[ 1 -lt 2 ]
echo $?
[ 10 -gt 1 ]
echo $?
[ 1 -eq 1 ]
echo $?
[ 1 -ne 1 ]
echo $?
```

Type the above code into the terminal in this lesson.

The binary operators used above are: `-lt` (less than), `-gt` (greater than), `-eq` (equals), and `-ne` (not equals). They deal happily with integers in single bracket tests, even when quoted.

`if` Statements

Now you understand tests, `if` statements will be easy!

Type this:

```
if [[ 10 -lt 2 ]]
then
  echo 'does not compute'
elif [[ 10 -gt 2 ]]
then
  echo 'computes'
else
  echo 'neither greater than, or less than, so must be equal'
fi
```

Type the above code into the terminal in this lesson.

`if` statements consist of:

- A test
- The word `then`
  - If the `if` test returned 'true' it runs the commands after the `then`
- If the `if` test returned false, it will
  - Drop to the next `elif` statement if there is another test
  - Or drop to the `else` block if there are no more tests
- Finally, the if block is closed with the `fi` string

The `else` or `elif` blocks are not required for the `if` statement to work. For example, this will also work:

```
if [[ 10 -lt 2 ]]
then
  echo 'does not compute'
fi
```

Type the above code into the terminal in this lesson.

Bare `if` Statements

It's easy to forget that the `if` statement in bash does not need angle brackets at all. If the code between the `then` and the `if` is a bash command, then it will trigger if the exit code of the command was 'true':

What will this output? No cheating!

```
if grep not_there /dev/null
then
    echo there
else
    echo not there
```

```
fi
```

Type the above code into the terminal in this lesson.

Line 1 has an `if` statement with a ‘bare’ `grep` command. If the exit code of that `grep` command is zero (ie a match was found) then it outputs `there`. Otherwise, it outputs `not there`.

Tests Quiz

1 Which of these will error if run?

COMPLETED 0%

1 of 2

What You Learned

We covered quite a lot in this lesson!

- What a *test* is in bash

- How to compare values within a *test*
- What the program `[` is
- How to perform logic operations with *tests*
- Some differences between `[` and `[[`
- The difference between *unary* and *binary operators*
- How types can matter in bash, and how to compare them
- `if` statements and *tests*

What Next?

Next you will cover another fundamental aspect of bash programming: **loops**.

Exercises

1) Research all the unary operators, and try using them (see `man bash`)

2) Write a script to check whether key files and directories are in their correct place.

3) Use the `find` and `wc` commands to count the number of files available in the lesson's terminal and perform different actions if the number is higher or lower than what you expect.