### The Shebang [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#the-shebang)

Run this:

1

2

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:

1

./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:

1

2

3

4

5

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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#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:

1

2

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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#the-path-variable)

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

1

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:

1

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?

1

2

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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#the-source-builtin)

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

1

2

3

4

5

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 exported

source simple\_echo                # source the script, which actually outputs the MYVAR variable





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 sourced 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 [#](https://www.educative.io/courses/master-the-bash-shell/qA7W9J4vnW7#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:

1

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:

**Command Substitution**

### Command Substitution Example [#](https://www.educative.io/courses/master-the-bash-shell/7n93Q4woE7r#command-substitution-example)

When writing bash scripts you often want to take the standard output of one command and ‘drop’ it into the script as though you had written that into it.

An example may help illustrate this idea. Type these commands:

1

2

3

hostname                            # 'hostname' outputs the name of the host

echo 'My hostname is: $(hostname)'  # Single quotes do not call the hostname command

echo "My hostname is: $(hostname)"  # Double quotes do, similar to variable dereferencing





Type the above code into the terminal in this lesson.

If those lines are placed in a script, it will output the hostname of the host the script is running on. This can make your script much more dynamic. You can set variables based on the output of commands, add debug, and so on, just as with any other programming language.

You may have noticed that if wrapped in single quotes, the special meaning of the $ sign is ignored again!

### The Two Command Substitution Methods [#](https://www.educative.io/courses/master-the-bash-shell/7n93Q4woE7r#the-two-command-substitution-methods)

There are two ways to do command substitution:

1

2

echo "My hostname is: `hostname`"   # backticks method

echo "My hostname is: $(hostname)"  # dollar-bracket method





Type the above code into the terminal in this lesson.

These give the same output and the backticks perform the same function. So which should you use?

#### The ‘Dollar-Bracket’ Method: $() [#](https://www.educative.io/courses/master-the-bash-shell/7n93Q4woE7r#the-dollar-bracket-method)

Type this:

1

2

3

4

5

mkdir tmp

cd tmp

echo $(touch $(ls ..))

cd -

rm -rf tmp





Type the above code into the terminal in this lesson.

What happened there?

* In **lines 1 and 2** you created and moved into a folder
* This temporary folder is cleaned up by **lines 4-5**
* **Line 3** is best read from the innermost parentheses outwards
  + The ls .. command is run in the innermost parentheses. This outputs the contents of the parent directory
  + This output is substituted in over the $(ls ..). The ‘words’ returned are placed as the arguments to the touch command. The touch command creates a set of empty files, based on the list of the parent directory’s contents
  + The echo command takes the output of the above substitution, in this case nothing, as touch does not produce any output

So, in summary:

* **Line 3** outputs the list of files of the parent directory
* Those filenames are also created locally as empty files

**Line 3** is an example of how subcommands can be nested. As you can see, the nesting is simple - just place a command wrapped inside a $() inside another command wrapped inside a $() and bash will then substitute the output of each command from the inside out for you in the appropriate order.

Now let’s look at the equivalent code with backticks.

#### The ‘Backtick’ Method [#](https://www.educative.io/courses/master-the-bash-shell/7n93Q4woE7r#the-backtick-method)

Type this out:

1

2

echo `touch \`ls ..\``

cd ..





Type the above code into the terminal in this lesson.

To nest the **backtick** version, you have to escape the inner backtick with a backslash, so bash knows which level the backtick should be interpreted at.

To demonstrate the difficulty here, a simple example suffices:

1

2

3

echo $(echo hello1 $(echo hello2))   # Nesting echoes

echo `echo hello1 `echo hello2``     # Nesting echoes with backticks fails

echo `echo hello1 \`echo hello2\``   # Nesting echoes with backticks and backslashes ok





Type the above code into the terminal in this lesson.

For historical reasons, the backtick form is still very popular, but I prefer the $() form because of the simplicity of managing nesting. You need to be aware of both, though, if you are looking at others’ code!

If you want to see how messy things can get, compare these two lines:

1

2

echo `echo \`echo \\\`echo inside\\\`\``

echo $(echo $(echo $(echo inside)))





Type the above code into the terminal in this lesson.

and consider which one is easier to read (and write)!

**Tests**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#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? [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#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

2

3

4

[ 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

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:

1

2

3

4

5

6

7

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? [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#what-is-really)

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

1

2

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:

1

2

3

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 [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#logical-operators)

What do you expect the output of this to be?

1

2

( [ 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

2

[  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 [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#the-operator)

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

1

2

[[ 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:

1

2

3

4

5

6

7

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 [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#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:

1

2

3

4

5

6

7

8

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:

1

2

3

4

5

6

7

8

9

10

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 [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#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:

1

2

3

4

5

6

7

8

[ 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:

1

2

3

4

5

6

7

8

9

10

11

12

[ 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 [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#if-statements)

Now you understand tests, if statements will be easy!

Type this:

1

2

3

4

5

6

7

8

9

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:

1

2

3

4

if [[ 10 -lt 2 ]]

then

  echo 'does not compute'

fi





Type the above code into the terminal in this lesson.

### Bare if Statements [#](https://www.educative.io/courses/master-the-bash-shell/392YWp2pOv4#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!

1

2

3

4

5

6

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.