**Loops**

### for Loops [#](https://www.educative.io/courses/master-the-bash-shell/B8VVZ908EDJ#for-loops)

First you’re going to run a for loop in a ‘traditional’ way:

1

2

3

4

5

6

    for (( i=0; i < 20; i++ ))

    do

      echo $i

      echo $i > file${i}.txt

    done

    ls





Type the above code into the terminal in this lesson.

* You just created twenty files, each with a number in them using a for loop in the ‘C’ language style (**line 1**)
* Note there’s no $ sign involved in the variable when it’s in the double parentheses!
* **Line 3** echoes the value of i in each iteration
* **line 4** redirects that value to a file called file{$i}.txt, where ${i} is replaced with its value in that iteration
* **Line 6** shows the listing of files created

1

2

3

4

for f in $(ls \*txt)

do

    echo "File $f contains: $(cat $f)"

done





Type the above code into the terminal in this lesson.

It’s our old friend the command substitution! The command substitution lists all the files we have.

This for loop uses the in keyword to separate the variable each iteration will assign to f and the list to take items from. Here bash evaluates the output of the ls command and uses that as the list, but we could have written something like:

1

2

3

4

5

for f in file1.txt file2.txt file3.txt

do

    echo "File $f contains: $(cat $f)"

done





Type the above code into the terminal in this lesson.

with a similar effect.

### while Loops in Bash [#](https://www.educative.io/courses/master-the-bash-shell/B8VVZ908EDJ#while-loops-in-bash)

While loops also exist in bash. Try and work out what’s going on in this trivial example:

1

2

3

4

5

6

7

8

9

10

n=0

while [[ ! -a newfile ]]

do

    ((n++))

    echo "In iteration $n"

    if [[ $(cat file${n}.txt) == "15" ]]

    then

            touch newfile

    fi

done





Type the above code into the terminal in this lesson.

* **Line 1** creates an n variable
* **Line 2-3** creates a while loop that increments the n variable on each iteration (**line 4**)
* **Line 5** output which iteration the loop is on
* If the file with the name file${n}.txt (where the ${n} is replaced by the value of the variable n at the time) contains the string 15 (**line 6**), then a newfile file is created
* The while loop condition on **line 2** finishes when this newfile file exists.

I often use while loops in the following ‘infinite loop’ form when running quick scripts on the command line:

1

2

3

4

5

6

7

8

9

10

11

n=0

while true

do

    sleep 1

    ((n++))

    echo $n seconds have passed

    if [[ $n -eq 60 ]]           # When n reaches 60...

    then

        break                    # Break out of the while loop

    fi

done





Type the above code into the terminal in this lesson.

The above code is an ‘infinite loop’ because the while condition is true, so will never break out by itself. The break statement on **line 11**, if reached, will exit the while loop.

### case Statements [#](https://www.educative.io/courses/master-the-bash-shell/B8VVZ908EDJ#case-statements)

Case statements may also be familiar from other languages. In bash, they’re most frequently used when processing command-line arguments within a script.

Before you look at a realistic case statement, type in this trivial one:

1

2

3

4

5

6

a=1                                        # Initialize 'a' variable

case "$a" in                               # Start case statement

1) echo 'a is 1'; echo 'ok1';;             # If a == 1, then output ok1

2) echo 'a is 2'; echo 'ok2';;             # If a == 2, then output ok2

\*) echo 'a is unmatched'; echo 'failure';; # If nothing was matched, output 'failure'

esac                                       # End case statement





Type the above code into the terminal in this lesson.

Try triggering the a is 2 case, and the a is unmatched case.

There are a few of new bits of syntax you may not have seen before.

* The double semi-colons ;; indicate that the next matching case is coming (rather than just another statement, as indicated by a single semi-colon)
* The 1) indicates what the case value $a should match. These values follow the globbing rules (so \* will match anything)
  + Try adding quotes around the values, or glob values, or matching a longer string with spaces
* The esac indicates the case statement is finished

### case Statements and Command Line Options [#](https://www.educative.io/courses/master-the-bash-shell/B8VVZ908EDJ#case-statements-and-command-line-options)

case statements are most often seen in the context of processing command-line options within shell scripts. There is a helper builtin just for this purpose: getopts.

Now you will write a more realistic example, and more like what is seen in ‘real’ shell scripts that uses getopts.

Create a file case.sh to try out a case statement with getopts:

1

2

3

4

5

6

7

8

9

10

11

12

cat > case.sh << 'EOF'

#!/bin/bash

while getopts "ab:c" opt

do

    case "$opt" in

    a) echo '-a invoked';;

    b) echo "-b invoked with argument: ${OPTARG}";;

    c) echo '-c invoked';;

    esac

done

EOF

chmod +x case.sh





Type the above code into the terminal in this lesson.

* **Line 1** creates a file case.sh using a here doc (see [here](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6302089980411904/draft) if you want to learn about these now)
* **Line 2** tells the system that this is a bash script
* **Line 3** Starts the while loop, calling the builtin getopts. The builtin returns true while there are more flags to process, and sets the opt variable to the flag being processed on this iteration. The string ab:c is the specification of the flags, where the : indicates that the previous flag mentioned takes an argument
* **Line 5** Starts a case statement based on the opt value set in this iteration
* **Line 6** deals with the -a flag. Note that the - sign is removed by getopts when passing into opt
* **Line 7** deals with the -b flag, which expects an argument. The argument is placed in the OPTARG variable by getopts
* **Line 8** deals with the -a flag

Run the above with various combinations and try and understand what’s happening:

1

2

3

4

5

./case.sh -a

./case.sh -b

./case.sh -b "an argument"

./case.sh -a -b -c

./case.sh





Type the above code into the terminal in this lesson.

**Exit Codes**

### What Is An Exit Code? [#](https://www.educative.io/courses/master-the-bash-shell/Y5gyoZV9g2p#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:

1

2

3

4

ls

echo $?

doesnotexist

echo $?





Type the above code into the terminal in this lesson.

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 [#](https://www.educative.io/courses/master-the-bash-shell/Y5gyoZV9g2p#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](https://www.educative.io/collection/page/5164406595911680/5419374779301888/5532163069968384)

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

### Exit Codes and if Statements [#](https://www.educative.io/courses/master-the-bash-shell/Y5gyoZV9g2p#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:

1

2

3

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:

1

2

3

4

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 [#](https://www.educative.io/courses/master-the-bash-shell/Y5gyoZV9g2p#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.

1

2

3

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:

1

2

3

4

5

6

7

8

9

10

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 [#](https://www.educative.io/courses/master-the-bash-shell/Y5gyoZV9g2p#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:

1

2

3

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.

**The 'set' Builtin**

### How Important is this Lesson? [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#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 [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#running-set)

Start by running set on its own:

1

set





Type the above code into the terminal in this lesson.

his 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](https://en.wikipedia.org/wiki/Standardization) specified by the [IEEE Computer Society](https://en.wikipedia.org/wiki/IEEE_Computer_Society) for maintaining compatibility between [operating systems](https://en.wikipedia.org/wiki/Operating_system).

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

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

1

2

3

4

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:

1

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 [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#set-vs-env)

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

1

2

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 [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#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:

1

2

3

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:

1

2

3

4

5

6

7

8

9

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:

1

2

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 [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#flags-with-set-instead-of-names)

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

1

2

set -e

set -x





Type the above code into the terminal in this lesson.

is the same as:

1

2

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 [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#the-pipefail-option-exit-codes-and-pipelines)

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

1

2

3

4

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.

1

2

3

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 [#](https://www.educative.io/courses/master-the-bash-shell/7nOMO9GBQDA#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:

1

2

shopt -s globstar

ls \*\*





Type the above code into the terminal in this lesson.