### Basic Functions [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#basic-functions)

Start by creating a simple function:

1

2

3

4

function myfunc {

    echo Hello World

}

myfunc





Type the above code into the terminal in this lesson.

By declaring a function, and placing the block of code that needs to run inside the curly braces, you can call that function on the command line as though it were a program.

### Arguments [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#arguments)

Unlike other languages, in bash there is no checking of functions’ arguments.

Predict the output of this, and then run it:

1

2

3

4

5

6

function myfunc {

    echo $1

    echo $2

}

myfunc "Hello World"

myfunc Hello World





Type the above code into the terminal in this lesson.

Can you explain the output? If not, you may want to re-read the previous lessons!

Arguments to functions are numbered, from **1 to n**. It’s up to the function to manage these arguments.

In the above example, in **line 5**, the quotes turn the Hello World string into a single argument, which is echoed on a single line. It is then followed by an empty line, because there was no second argument provided. In **line 6**, the Hello World string is not enclosed in quotes, so is treated as two separate variables, which are echoed on two separate lines as two separate arguments.

This illustrates a very important point about shells, which is that they separate items by space by default. This will become very important when we discuss the IFS in a [later lesson](https://www.educative.io/collection/page/5164406595911680/5419374779301888/6615375431073792)

### Variable Scope [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#variable-scope)

Variables can have scope in bash. This is particularly useful in functions, where you don’t want your variables to be accessible from outside the function.

These commands illustrate this:

1

2

3

4

5

6

function myfunc {

    echo $myvar

}

myfunc

myvar="Hi from outside the function"

myfunc





Type the above code into the terminal in this lesson.

Bash functions have no special scope. Variables outside are visible to it.

There is, however, the capability within bash to declare a variable as local:

1

2

3

4

5

6

7

function myfunc {

    local myvar="Hi from inside the function"   # Creates a variable local to the function

    echo $myvar

}

myfunc

echo $myvar

local myvar="Will this work?"                   # Will not work





Type the above code into the terminal in this lesson.

The variable declared with local can only be viewed and accessed within the function, hence, it doesn’t interfere with the code outside the function. As seen from the example above, it can’t be declared outside a function.

The local above is an example of a bash ‘builtin’. Now is a good time to talk about the different types of commands.

### Functions, Builtins, Aliases and Programs [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#functions-builtins-aliases-and-programs)

There are at least four ways to call commands in bash:

* Builtins
* Functions
* Programs
* Aliases

Let’s take a look at each type of command one by one.

### Builtins [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#builtins)

**Builtins** are commands that come ‘built in’ to the bash shell program. Normally you can’t easily tell the difference between a builtin, a program or a function, but after reading this you will be able to.

The most commonly-used builtin is cd. There is also one called builtin!

1

2

3

4

builtin cd /tmp        # Call cd as normal with the builtin builtin

cd -

builtin grep           # Is 'grep' a builtin?

builtin notaprogram    # What happens if the command doesn't exist at all





Type the above code into the terminal in this lesson.

As you’ve probably guessed by typing the above commands in the terminal, the builtin builtin calls the builtin program (this time cd), and throws an error if no such builtin exists.

In case you didn’t know, cd - returns you to the previous directory you were in.

### Functions [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#functions)

Functions we have covered above, but what happens if we write a function that clashes with a builtin?

What if you create a function called cd?

1

2

3

4

5

6

7

8

9

function cd() {

    echo 'No!'

}

cd /tmp

builtin cd /tmp

cd -

unset -f cd

cd /tmp

cd -





Type the above code into the terminal in this lesson.

* In **lines 1-3** you created a function called cd that just outputs No!
* In **line 4** you then tried to move to another folder, and saw that the function handled the cd command over the built-in cd command
* In **line 5** you explicitly called the cd built-in using the builtin command, which ensures that the builtin is called over a function or a program
* After returning to the previous folder with cd - in **line 6** you unset the function with the unset -f built-in command in **line 7**, after which cd can be used as normal in **lines 8 and 9**

Now type this in:

1

2

3

4

5

6

function cd() {

    echo 'Function cd'

}

declare -f              # Show functions in the environment

declare -F              # Show just the function names

unset -f cd





Type the above code into the terminal in this lesson.

If you want to know what functions are set in your environment, you run declare -f. This will output the functions and their bodies. If you just want the names, use the -F flag.

### Programs [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#programs)

**Programs** are executable files. Commonly-used examples of these are programs such as grep, sed, vi, and so on.

How do you tell whether a command is a builtin or a separate binary?

First, see whether it’s a builtin by running builtin <command> as you did before. Then you can also run the which command to determine where the file is stored on your filesystem.

1

2

3

4

which grep               # Where is the grep program?

which cd

which builtin

which doesnotexist





Type the above code into the terminal in this lesson.

### Aliases [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#aliases)

Finally there are **aliases**. Aliases are strings that the shell takes and translates to whatever that string is aliased to.

Try this and explain to yourself what is going on as you go:

1

2

3

4

5

6

7

alias cd=doesnotexist

alias

cd

unalias cd

cd /tmp

cd -

alias





Type the above code into the terminal in this lesson.

* In **line 1** you alias the cd command to the doesnotexist command (which does not exist)
* **Line 2** shows the aliases available
* **line 3** shows that the alias has ‘taken over’ the cd command
* unalias in **line 4** removes the alias you created, so that
* **lines 5-6** cd can be used as normal again
* Running alias again in **line 7** shows that the alias for cd is no longer there

### The ‘type’ Builtin [#](https://www.educative.io/courses/master-the-bash-shell/g7yxEl6BvwG#the-type-builtin)

There is also a builtin called type that tells you how a command would be interpreted by the shell:

1

2

3

type ls       # What kind of command is 'ls'?

type pwd

type myfunc





Type the above code into the terminal in this lesson.

**Pipes and Redirects**

### Basic Redirects [#](https://www.educative.io/courses/master-the-bash-shell/3w5GrpYLENn#basic-redirects)

Start off by creating a file:

1

echo "contents of file1" > file1





Type the above code into the terminal in this lesson.

The > character is the redirect operator. This takes the output from the preceding command that you’d normally see in the terminal and sends it to a file that you give it. Here it creates a file called file1 and puts the echoed string into it. Try cating the file if you want to check by running:

cat file1

There’s a subtlety here which we’ll get to: sometimes not all the output you see in the terminal would get redirected by this, but don’t worry about this yet.

### Basic Pipes [#](https://www.educative.io/courses/master-the-bash-shell/3w5GrpYLENn#basic-pipes)

Now type this in:

1

cat file1 | grep -c file





Type the above code into the terminal in this lesson.

Note: If you don’t know what grep is, you will need to learn. This is a good place to start: <https://en.wikipedia.org/wiki/Grep>

Normally you’d run a grep with the filename as the last argument, but instead here we **pipe** the contents of file1 into the grep command by using the ‘pipe’ operator: |. The resulting output of 1 indicates the number lines that matched the word file in the the file called file1.

A pipe takes the standard output of one command and passes it as the input to another. What, then is **standard output**, really? You will find out soon!

1

cat file2





Type the above code into the terminal in this lesson.

You should have seen an error, because the file did not exist.

Now run this, and try and guess the result before you run it:

1

cat file2 | grep -c file





Type the above code into the terminal in this lesson.

You should have seen two lines. The first will be an error

cat: file2: No such file or directory

and the second line a 0.

We will explain further below why a 0 is shown, even though a line was outputted that had the word file in it.

Note: See Exercise 3 below for an often-useful operator that changes this behaviour.

### Standard Output vs Standard Error [#](https://www.educative.io/courses/master-the-bash-shell/3w5GrpYLENn#standard-output-vs-standard-error)

In addition to **standard output**, there is also a **standard error** channel. When you pass a non-existent file to the cat command, it throws an error message out to the terminal, as you would have seen when you attempted the code above. Although the message looks the same as the contents of a file, it is sent to a different output channel. In this case it’s standard error rather than standard output.

As a result, it is NOT passed through the pipe to the grep command, and grep counts zero matches in its output.

To the viewer of the terminal, there is no difference, but to bash there is all the difference in the world!

There is a simpler way to refer to these channels. A number is assigned to each of them by the operating system.

These are the numbered file descriptors, and the first three are assigned to the numbers 0,1 and 2.

* 0 is standard input
* 1 is standard output
* 2 is standard error

When you redirect ‘standard output’ to a file, you use the redirection operator >. Implicitly, you are using the 1 file descriptor.

Type this to see an example of redirecting 2, which is ‘standard error’.

1

2

command\_does\_not\_exist

command\_does\_not\_exist 2> /dev/null





Type the above code into the terminal in this lesson.

* **Line 1** above will show an error as you might expect, but **line 2** does not
* In **line 2**, the file descriptor 2 (standard error) is directed to a file called /dev/null

The file /dev/null is a special file created by Linux (and UNIX) kernels. It is effectively a black hole into which data can be pumped: anything written to it will be absorbed and ignored.

Another commonly seen redirection operator is 2>&1:

1

command\_does\_not\_exist 2>&1





Type the above code into the terminal in this lesson.

What this does is tell the shell to send the output on standard error (2) to whatever endpoint standard output is pointed to at that point in the command.

Since standard output is pointed at the terminal at that time, standard error is also pointed at the terminal. From your point of view you see no difference, since both standard output and standard error are pointed at the terminal anyway.

But when we try and redirect to standard error or standard output to files things get interesting, as you can change where they go. You saw this above when we redirected standard error to /dev/null.

Now type these in and try and figure out why they produce different output:

1

2

3

4

command\_does\_not\_exist 2>&1 > outfile

cat outfile

command\_does\_not\_exist > outfile 2>&1

cat outfile





Type the above code into the terminal in this lesson.

This is where things get tricky and you need to think carefully!

* **Line 1** will have displayed the error as output and put nothing into the redirected-to outfile (**line 2**), while
* **Line 3** did not display the error and captured it in the outfile (**line 4**).

Remember that the redirection operator 2>&1 points standard error (file descriptor 2) at whatever standard output (file descriptor 1) was pointed to at the time.

If you read the first line carefully, at the point 2>&1 was used, standard output was pointed at the terminal. So standard error is pointed at the terminal from there on.

After that point, standard output is redirected (with the > operator) to the file outfile.

So at the end of all this:

* The standard error of the output of the command command\_does\_not\_exist points at the terminal
* The standard output points at the file outfile.

In the second line (command\_does\_not\_exist > outfile 2>&1), what is different?

The order of redirections is changed.

Now:

* The standard output of the command command\_does\_not\_exist is pointed at the file outfile
* The redirection operator 2>&1 points file descriptor 2 (standard error) to whatever file descriptor 1 (standard output) is pointed at

So in effect, both standard out and standard error are pointed at the same file (outfile).

This pattern of sending all the output to a single file is seen very often, and few understand why it has to be in that order. Once you understand, you will never pause to think about which way round the operators should go again!

### Differences Between Pipes and Redirects [#](https://www.educative.io/courses/master-the-bash-shell/3w5GrpYLENn#differences-between-pipes-and-redirects)

To recap:

* A pipe passes standard output as the standard input to another command
* A redirect sends a channel of output to a file

A couple of other commonly used operators are worth mentioning here:

1

grep -c file < file1





Type the above code into the terminal in this lesson.

The < operator redirects standard input to the grep command from a file. In this case, it is equivalent to the cat file1 | grep -c file command you saw earlier.

1

2

3

4

5

6

echo line1 > file3

cat file3

echo line2 > file3   # Overwrites file3, replacing its contents with 'line2'

cat file3

echo line3 >> file3  # Appends to file3

cat file3





Type the above code into the terminal in this lesson.

* **Lines 1 and 3** above use the > operator, while
* **Line 5** uses the >> operator.

The > operator effectively creates the file anew whether it already exists or not. The >> operator, by contrast, appends to the end of the file.

As a result, only line2 and line3 are added to file3.