Functions

This lesson discusses the four types of a ‘command’ that exist in bash: functions, aliases, programs, and built-ins.

WE'LL COVER THE FOLLOWING

From one angle, bash can be viewed as a programming language, albeit a quite slow and primitive one.

One of the language features it has is the capability to create and call *functions*.

This leads us onto the topic of what a ‘command’ can be in bash, and we cover all four of them: *functions*, *aliases*, *programs*, and *builtins*.

By the end you will have a more nuanced understanding of commands in bash.

How Important is this Lesson?

It's possible to get by without knowing much about functions in bash, but any serious bash user will know what they do and how they work.

Basic Functions

Start by creating a simple function:

```
function myfunc {
    echo Hello World
}
myfunc
```

Type the above code into the terminal in this lesson.

Terminal

> Note: continue to use this terminal during this lesson. Throughout this course it is assumed that you complete each lesson by typing in the commands in the provided terminal for that lesson in the order the commands are given.

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

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

Predict the output of this, and then run it:

```
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

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 `echo`ed 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

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:

```
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:

```
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

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

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`!

```
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

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`?

```
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:

```
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

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.

```
which grep                  # Where is the grep program?
which cd
which builtin
which doesnotexist
```

Type the above code into the terminal in this lesson.

Is `which` a builtin or a program?

Show Hint

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:

```
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

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

```
type ls         # What kind of command is 'ls'?
type pwd
type myfunc
```

Type the above code into the terminal in this lesson.

Functions Quiz

1 Choose the true statements

COMPLETED 0%

1 of 2

What You Learned

- Basic function creation in bash
- Functions and variable scope
- Differences between functions, builtins, aliases and programs

What Next?

Next you will learn about **pipes and redirects** in bash. Once learned, you will have all you need to get to writing shell scripts in earnest.

Exercises

1) Run `typeset -f`. Find out how this relates to `declare -f` by looking at the bash man page (`man bash`).

2) alias `alias`, override `cd`. Try and break things. Have fun. If you get stuck, close down your terminal, or exit your bash shell (if you haven't overridden exit!).