**Tutorial 5**

**Shell Scripting**

Scripts are collections of commands that are stored in a file. The shell can read this file and act on the commands as if they were typed at the keyboard. They are extremely useful for automating tasks on [Linux](http://www.linfo.org/linuxdef.html) and other [Unix-like](http://www.linfo.org/unix-like.html) [operating systems](http://www.linfo.org/operating_systems_list.html).

Each shell script consists of:

* **Shell keywords** such as if..else, do..while.
* **Shell commands** such as pwd, test, echo, continue, type.
* **Linux binary commands** such as w, who, free etc.
* **Text processing utilities** such as grep, awk, cut.
* **Functions** - add frequent actions together via functions. For example, /etc/init.d/functions file contains functions to be used by most or all system shell scripts in the /etc/init.d directory.
* **Control flow** statements such as if..then..else or shell loops to perform repeated actions.

Using a text editor enter the following:

#!/bin/bash

echo Hello World

# This is a comment

The first line tells Unix that the file is to be executed by /bin/bash. This is the standard location of the Bourne Again shell on just about every Unix/Linux system. The third line begins with a special symbol: #. This marks the line as a comment, and it is ignored completely by the shell.

The only exception is when the *very first* line of the file starts with **#!** - as ours does. This is a special directive which Unix treats specially. It means that even if you are using **csh, ksh**, or anything else as your interactive shell, that what follows should be interpreted by the **bash shell**. The third line runs a command: echo, with two parameters, or arguments - the first is "Hello"; the second is "World".

Note that echo will automatically put a single space between its parameters. The # symbol still marks a comment; the # and anything following it is ignored by the shell.

Now run chmod 755 first to make the text file executable, and run *./first*. The *./* tells the interpreter to run the program from the current directory. Your screen should then look like this:

> chmod 755 first

> ./first

Hello World $

**Quotes**

Re-enter the editor and add spaces between the Hello and World. **What happens?**

Now try:

#!/bin/bash

echo "Hello World"

# This is a comment

*What happens this time?*

The reason for this is because echo has now been called with just ONE argument - the string "Hello It prints this out exactly. Note that the shell parses the arguments BEFORE passing them on to the program being called. In this case, it strips the quotes but passes the string as one argument.

Single and double quotes can also be used for dealing with spaces. The difference between single and double quotes being that in double quotes the **$**, **\**, and **'** characters still preserve their special meanings. Single quotes will take the **$** and **\** literally and regard the **'** as the end of the string. Here's an example:

> VAR='This is my text'

> echo $VAR This is my text

This is my text This is my text

> echo "$VAR"

This is my text

> echo '$VAR' $VAR

The string following the **$** character is interpreted as being a variable except when enclosed in single quotes as shown above.

**Command Substitution**

Back quotes are used for command substitution e.g

> echo Today is date

Today is date

> echo Today is `date`

It will print today's date as, Today is Fri October 28th 2016. Notice that the `date` statement uses back quotes.

Table 2 Summary of uses of quotes

|  |  |  |
| --- | --- | --- |
| **Quotes** | **Name** | **Meaning** |
| **"** | Double Quotes | "Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $). |
| **'** | Single quotes | 'Single quotes' - Enclosed in single quotes remains unchanged. |
| **`** | Back quote | `Back quote` - To execute command |

However in my estimation it is probably better to use *$(command),* rather than the back quotes as they are easily confused with normal quotes.

e.g

echo Today is $(date)

Will also print today's date as, Today is Fri October 28th 2016.

**Escape Characters**

Escape characters are used to remove the special meaning from a single character. A non-quoted backslash, \, is used as an escape character in Bash. It preserves the literal value of the next character that follows, with the exception of *newline*. If a newline character appears immediately after the backslash, it marks the continuation of a line when it is longer that the width of the terminal; the backslash is removed from the input stream and effectively ignored.

> fred**=***20021226*

> echo $fred 20021226

20021226 20021226

> echo *\$fred*

$fred

and another example to display: Hello “World”

>echo "Hello \"World\""

Hello "World"

**Writing Shell Scripts**

Use gedit to open a file giving it the name of your choice. Then enter the following scripts. However, before you can run them you will have to change the permissions using chmod and then to run them you will need to precede the filename with *./* e.g. *./fred* will run the file *fred* in your current directory.

1.

#!/bin/bash

STR=”Hello World”

echo $STR

2.

#!/bin/bash

#This script displays the date, time, username

# and current directory

echo date and time is :

date

echo

echo Your usename is : `whoami`

echo Your current directory is :

pwd

3.

#!/bin/bash

touch filea fileb filec

echo Type in a line of text

read x

home=`pwd`

echo $home

echo $x >> /home/filea

**Note** The *touch* [command](http://www.linfo.org/command.html) is the easiest way to create new, empty [files](http://www.linfo.org/file.html). It is also used to change the *timestamps* (i.e., dates and times of the most recent access and modification) on existing files and [directories](http://www.linfo.org/directory.html). - touch's syntax is:

touch [option] file\_name(s)

**Creating a personal bin directory**

Up till now, to execute scripts you have entered the absolute path name e.g. *./fred .*

However, it is more convenient to create a personal bin directory for all your scripts. Firstly, change to your home directory and create a subdirectory named bin.

> cd ~

> mkdir bin

A little later you will need to know the absolute pathname to this directory. To do this cd into the bin directory and use the pwd command. e.g.

> cd ~/bin

> pwd

Now add bin to your path (*not required now at Northampton as the technician has already created the path*). To do this you will need to edit your shell startup file called *.bash\_profile*.

Now use gedit to *open .bash\_profile*, and do a save as *bash\_profile\_backup* to make a backup of the original, then add the following line to the end of the file.

PATH=${PATH}:*other paths*:*absolute path name to your bin directory*

e.g.

PATH=$PATH:$HOME/bin

#where $HOME is a variable containing the path to your home directory

export PATH

Or alternatively

PATH=$PATH:/home/staff/brian/bin/bin

export PATH

You will need to run .bash\_profile – usually you need to restart Centos, but it can be run by typing:

source ~/.bash\_profile

You can now run scripts from bin without the need for a ./.

**Arithmetic**

The simplest way is to surround the sum with $((..)). e.g.

x = $(( 3 + 4))

echo $x

Alternatively you can use expr, try the following

> expr 2 + 1

> expr 7 – 2

> expr 4 \* 5

The last one produces an error message as the multiplication operator needs to be quoted (preceded by a \ ) to protect it from shell expansion. This is because the \* is a metacharacter, which have a special meaning to the shell, such as **< > \* ? | &.**

> expr 8 / 5

> expr 19 % 5

> expr 6 + 7 \\* 4

> expr `expr 6 + 7` \\* 4

***bc - the basic calculator***

The bc utility is a programmable calculator. It allows several types of calculations and provides simple looping logic. It is also much easier to read than the expr expression evaluator.

To start bc, type bc at the command line. Once bc begins, you are using calculator until you type ***quit***. Try the following example.

> bc

4+2

6

quit

>

***Using variables in bc***

Some versions of bc allow only single character variables, though later versions allow multi-character variable names. Notice that the assignments do not result in any output from bc, it requires you to type an expression or variable without an assignment for it to be output to the screen.

> bc

a=4

b=5

c=b-a

c

1

quit

>

The dot variable holds the last output. In the following example the user enters 4+2 and bc outputs 6. Then the user enters . (dot) and bc outputs 6, the value held in the temporary variable, again. The variable can itself be used in an expression as in .+1, and bc outputs 7. Finally typing . (dot) will again output 7 because the last output value, 7, has been assigned to . (dot).

> bc

4+2

6

.

6

.+1

7

.

7

quit

>

***Using bc in shell scripts***

To use bc in shell scripts just write a string with echo and pipe it into the input of bc.

To add two floating point numbers say 2.5 and 3.75, then type the string in quotes *“2.5+3.75″* pipe it into the input of bc :

echo "2.5+3.75" | bc

6.25

The same can be done for subtraction, division and multiplication.

The scale variable determines the number of digits which follow the decimal point in your result. By default, the value of the scale variable is zero. This can be set by declaring scale before your calculation, as in the following division example:

> echo “scale=25;57/43” | bc

1.3255813953488372093023255

***Square roots***

> echo “scale=30;sqrt(2)” | bc

1.414213562373095048801688724209

***Powers***

> echo “6^6” | bc

46656

The use of brackets. If you try this:

> echo “6^6^6” | bc

You will either get a screen full of numbers or an error message. You need to type:

> echo “(6^6)^6” | bc

Whereas what you did type was interpreted as:

> echo “6^(6^6)” | bc

Example shell script

#!/bin/bash

echo "Enter a floating point number: "

read a

echo "Enter another floating point number: "

read b

echo "Addition: "

echo "$a+$b" | bc

**To assign the output from bc to a variable**

x=$(echo “$a+$b”|bc )