Introduction to writing shell scripts

using BASH (the Bourne-Again Shell)

BASH is named after the Bourne Shell of 1977 and incorporates features of the C Shell and the Korn Shell (alternative command shells to using BASH)

BASH is provided as the command shell on Mac OS X and is usually the default command shell on Linux distros.

Suggested GUI editors for shell scripts

`gedit` (provided as standard with many distros)

`kwrite` (provided as standard with KDE distros)

`Geany` is a good programmers' editor for those new to writing scripts

or any other programmers' editor

All provide syntax highlighting

Programmers' editors provide better facilities

Suggested command line editors for shell scripts

`pico` / **`nano`** - provided by most distros

`mcedit` - part of the Midnight Commander (**`mc`**) package

or anything else you know and like!

Most provide syntax highlighting

Traditional Unix editors like **`vi`**, **`vim`** and **`emacs`** have extensive facilities (and alternative GUI interfaces) but have a steep learning curve

Running a shell script

Either give script name as an argument to the interpreter:

```
bash /path/to/script
```

Or name the script interpreter program in line one of the script and make the script executable so it becomes a command you can run:

line 1 of script:

```
#!/bin/bash
```

set executable:

`chmod +x /path/to/script` *(or set executable using GUI tool)*

There can be some subtle differences.

Running an executable script

You can always use

```
/full/path/to/myscript
```

If the script is in a folder on your PATH you can use

```
myscript
```

If the script is in the current working directory you can use

```
./myscript
```

How Shell Scripts are interpreted

Lines are processed one at a time

any variables on the line are expanded (ie actual values inserted)

output from “command substitution” is collected

the command(s) on the line are then executed

Command substitution:

the text output of running a command is collected and is inserted on the line replacing the command

eg `today=`date`` expands to

```
today="Wed Mar 30 12:23:24 BST 2016"
```

Pitfalls in writing scripts - spaces

You must know when, and when not, to use spaces!

Variable assignment:

`VAR1="some text"`	OK
`VAR1 = "some text"`	Space around = sign not allowed

Tests

`if [ $TYPE = "jpg" ]`	OK
`if [$TYPE = "jpg"]`	Space either side of [required

Pitfalls in writing scripts – quote marks

Double quote marks are needed when a number of words must be treated as a single string.

`MyFile="A filename with spaces.txt"`	OK
`MyFile=AFilenameWithNoSpaces.txt`	OK
`MyFile=A filename with spaces.txt`	Error!

Watch out for:

`MyFile="A filename with spaces.txt"`	This is OK, but
`ls -l $Myfile`	This is an error!

Why is it an error? Because it expands to ...

```
ls -l A filename with spaces.txt
```

… because the quote marks are not saved as part of the variable.

So you always need to use quotes if the variable may contain spaces: `ls -l "$Myfile"`

Pitfalls in writing scripts – error checking

Syntax errors are only picked up line by line as the script is run.

Testing needs to be "non-destructive".

If external commands fail (or are not found) this is not regarded as an error and the script will carry on.

Consider checking the result of running each external command.

After running an external command, the special shell variable **$?** contains the return code from the command.

Example

```
xxx   (some command which returns 1 for failure)
if [ $? -eq 1 ]
      then
      echo "Problem running program xxx"
      exit
fi
```

Commonly used commands in scripts

Print some text

echo

Assign value to a variable

VAR=something

Set the working folder (directory)

cd *absolute or relative path*

Quit the script at this point

exit *optionally with a return value*

Run commands under certain conditions

if [*some test* **]**

then *commands to run if the test is satisfied*

fi

Alternative form (more compact but cryptic!)

test *some test* **||** *command to execute*

Example

test -w mylogfile || exit

Check mylogfile is writeable by me or exit if it isn't

Loop a fixed number of times

for VAR in *a space-separated list*

do *some command*

done

Loop an unknown number of times

while [*some test is true* **]**

do *some command*

done

(There are other loop mechanisms too.)

Redirect program output somewhere other than the screen

>

Example

echo "Log of script execution" > mylogfile

Create the file if it doesn't exist, or overwrite it if it does

Redirect output, but appending to an existing file

>>

Example

echo "Log of script execution" >> mylogfile

Will still create the file if it doesn't exist

Grouping commands

{

commands

more commands

}

Comments

#

Can be at the start of line or anywhere within a line

Examples

```
# a whole line comment
webdir='/var/www/html'   # a line-end comment
```

Defining a function *(there are alternate forms)*

```
function_name ()
{
commands
}
```

Functions must be defined before they can be used.

Either define functions early in a script, or include them near the start of your script from a separate file:

```
source /path/to/myfunctions.file
```

Frequently used external commands

Searching for files that meet particular criteria

`find`

Example

```
find -name '*.mp4' -size +1G
```

Searching for text within files

`grep`

Example

```
grep 'memory_limit' php.ini
```

Manipulating lines of data

`cut`

Example

```
cut -d',' -f3 somefile.csv
```

extract the third field from a comma-separated file

Editing text within files

`sed`

Example

```
sed -i 's/green/blue/g' somefile.txt
```

Displaying a GUI status message from within a script

`notify-send`

Example

```
notify-send "test status message"
```

notify-send may not be installed by default on all distros

Create GUI dialog from within a script

`zenity`

Example

```
zenity –question –text "Do you want to continue?"
```

zenity may not be installed by default on all distros