6BASH SHELL SCRIPTING

1. Definition

Shell

Shell is a macro processor, which allows you by use of commands to interact with your computer.

Scripting

Scripting allows for an automatic command execution that would otherwise be executed interactively one by one.

Bash

Bash is a command language interpreter. It is widely available on various operating systems and is a default command interpreter on most GNU/Linux systems. The name is an acronym for the 'Bourne-Again SHell'.

To see what is your default interpreter execute command `echo $SHELL`:

```
$ echo $SHELL
/bin/bash
```

To define your script's interpreter as Bash, first locate a full path to its executable binary using `which` command, prefix it with a shebang `#!` and insert it as the first line of your script.

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

Shebang

The #! syntax used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems. Most Linux shell and perl / python script starts with the following line:

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

2. File Names and Permissions

- In order to execute the shell script, the file needs to be made executable by use of `chmod +x FILENAME` command. By default, any newly created files are not executable regardless of their file extension suffix.
- The file extension on GNU/Linux systems mostly does not have any meaning.
- On GNU/Linux systems a `file` command can be used to identify a type of the file.
- The file extension does not hold any value, and the shell interpreter, in the shell script case, carries more weight.

```
linuxconfig.org:~$  file hello-world.sh
hello-world.sh: Bourne-Again shell script, ASCII text executable
linuxconfig.org:~$ cp hello-world.sh 0_xvz
linuxconfig.org:~$ file 0_xvz
0_xvz: Bourne-Again shell script, ASCII text executable
linuxconfig.org:~$ vi 0_xvz
linuxconfig.org:~$ file 0_xvz
0_xvz: ASCII text
```

Thus, shell script name `0_xyz` is perfectly valid, but if possible it should be avoided.

3. Script Execution

In a highly simplistic view, a bash script is nothing else just a text file containing instructions to be executed in order from top to bottom. How the instructions are interpreted depends on defined shebang or the way the script is executed.

```
linuxconfig.org:~$ echo date > date.sh
linuxconfig.org:~$ cat date.sh
date
linuxconfig.org:~$ ./date.sh
bash: ./date.sh: Permission denied
linuxconfig.org:~$ bash date.sh
Thu 20 Jul 11:46:30 AEST 2017
linuxconfig.org:~$ vi date.sh
linuxconfig.org:~$ chmod +x date.sh
linuxconfig.org:~$ ./date.sh
Thu 20 Jul 11:46:49 AEST 2017
```

Another way to execute bash scripts is to call bash interpreter explicitly eg. `$ bash date.sh`, hence executing the script without the need to make the shell script executable and without declaring shebang directly within a

shell script. By calling bash executable binary explicitly, the content of our file `date.sh` is loaded and interpreted as Bash Shell Script.

4. Hello World Bash Shell Script

- Make one file with the following code and .sh extension.

```
#!/bin/bash

echo "Hello World"
```

- Names it `hello-world.sh`
- make your script executable with the `chmod` command and execute it using relative path `./hello-world.sh`:

```
$ sudo chmod +x hello-world.sh
$linuxconfig.org:~$./hello-world.sh
Hello World
```

5.Simple Backup Bash Shell Script

Insert the following code into a new file called `backup.sh`, make the script executable, and run it:

```
#!/bin/bash

tar -czf /tmp/groovy.tar.gz /home/groovy
```

-czf:-

c(--create)=Create a new archive.

f(**--file=***ARCHIVE*)=Use archive file (or device) *ARCHIVE*.

z(**--gzip**, **--gunzip**) = This option tells **tar** to read or write archives through **gzip**, allowing **tar** to directly operate on several kinds of compressed archives transparently. This option should be used, for example, when operating on files with the extension **.tar.gz**.

Further information:- https://www.computerhope.com/unix/utar.html

6. Variables

Create a new script `welcome.sh` with the following content:

```
#!/bin/bash

greeting="Welcome"
user=$(whoami)
day=$(date +%A)

echo "$greeting back $user! Today is $day, which is the best day of the entire week!"
echo "Your Bash shell version is: $BASH_VERSION. Enjoy!"
```

The script uses a new shell scripting trick ${parameter} called parameter expansion. In our case, curly braces {} are required because our variable $user is followed by characters which are not part of its variable name.

7. Input, Output, and Error Redirections

Consider the following example:

```
linuxconfig.org:~$ ls foobar barfoo
ls: cannot access 'barfoo': No such file or directory
foobar
linuxconfig.org:~$ ls foobar barfoo > stdout.txt
ls: cannot access 'barfoo': No such file or directory
linuxconfig.org:~$ ls foobar barfoo 2> stderr.txt
foobar
linuxconfig.org:~$ ls foobar barfoo &> stdoutandstderr.txt
linuxconfig.org:~$ cat stdout.txt
foobar
linuxconfig.org:~$ cat stderr.txt
ls: cannot access 'barfoo': No such file or directory
linuxconfig.org:~$ cat stdoutandstderr.txt
ls: cannot access 'barfoo': No such file or directory
foobar
```

The `>` notation is used to redirect **stdout** to a file whereas `2>` notation is used to redirect **stderr** and `&>` is used to redirect both **stdout** and **stderr**. The `cat` command is used to display a content of any given file.

The alternative method is to accept command input from a file using `<` notation. Consider the following example.

```
linuxconfig.org:~$ cat > file1.txt
I am using keyboard to input text.
Cat command reads my keyboard input, converts it to stdout which
is instantly redirected to file1.txt
That is, until I press CTRL+D
linuxconfig.org:~$ cat < file1.txt
I am using keyboard to input text.
Cat command reads my keyboard input, converts it to stdout which
is instantly redirected to file1.txt
That is, until I press CTRL+D
```

8. Functions

```
linuxconfig.org:~$ vi function.sh
linuxconfig.org:~$ chmod +x function.sh
linuxconfig.org:~$ ./function.sh
User Name: linuxconfig
Home Directory: /home/linuxconfig
linuxconfig.org:~$ vi function.sh
linuxconfig.org:~$ ./function.sh
User Name: linuxconfig
Home Directory: /home/linuxconfig
User Name: linuxconfig
Home Directory: /home/linuxconfig
```

```
#!/bin/bash

function user_details {
        echo "User Name: $(whoami)"
        echo "Home Directory: $HOME"
}

user_details
user_details
```

One more example:

After reviewing the above backup.sh script, you will notice the following changes to the code:

- we have defined a new function called `total_files`. The function utilized the `find` and `wc` commands to determine the number of files located within a directory supplied to it during the function call
- we have defined a new function called `total_directories`. Same as the above `total_files` function it utilized the `find` and `wc` commands however it reports a number of directories within a directory supplied to it during the function call
- output

```
#!/bin/bash

# This bash script is used to backup a user's home directory to /tmp/.

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

# The function total_files reports a total number of files for a given directory.
function total_files {
        find $1 -type f | wc -l
}

# The function total_directories reports a total number of directories
# for a given directory.
function total_directories {
        find $1 -type d | wc -l
}

tar -czf $output $input 2> /dev/null

echo -n "Files to be included:"
total_files $input
echo -n "Directories to be included:"
total_directories $input

echo "Backup of $input completed!"

echo "Details about the output backup file:"
ls -l $output
```

```
$ ./backup.sh
Files to be included:19
Directories to be inlcuded:2
Backup of /home/linuxconfig completed!
Details about the output backup file:

-rw-r--r-- 1 linuxconfig linuxconfig 5520 Aug 16 11:01
/tmp/linuxconfig_home_2017-08-16_110121.tar.gz
```

9. Numeric and String Comparison

The following table lists rudimentary comparison operators for both numbers and strings:

Description	Numeric Comparison	String Comparison
less than	-lt	<
greater than	-gt	>
equal	-eq	=
not equal	-ne	!=
less or equal	-le	N/A
greater or equal	-ge	N/A
Shell comparison example:	*[100 -eq 50]; echo $?*	*["GNU" = "UNIX"]; echo $?*

echo $? command,check for a return value of the previously executed evaluation.

```
linuxconfig.org:~$ a=1
linuxconfig.org:~$ b=2
linuxconfig.org:~$ [ $a -lt $b ]
linuxconfig.org:~$ echo $?
0
linuxconfig.org:~$ [ $a -gt $b ]
linuxconfig.org:~$ echo $?
1
linuxconfig.org:~$ [ $a -eq $b ]
linuxconfig.org:~$ echo $?
1
linuxconfig.org:~$ [ $a -ne $b ]
linuxconfig.org:~$ echo $?
0
```

Using string comparison operators we can also compare strings in the same manner as when comparing numeric values. Consider the following example:

```
linuxconfig.org:~$ [ "apples" = "oranges" ]
linuxconfig.org:~$ echo $?
1
linuxconfig.org:~$ str1="apples"
linuxconfig.org:~$ str2="oranges"
linuxconfig.org:~$ [ $str1 = $str2 ]
linuxconfig.org:~$ echo $?
1
```