Shell Scripting – Class Session 1

1. Basic Introduction

When you work with Shell Scripts in UNIX or Linux, you always must consider the SHELL that you are using whether it's a BashShell, a KornShell, a C Shell and so on. The reason is because the syntax is a little different in the handling of variables and functions etc. As part of examples used as part of this document, we use BashShell. First, to verify the shell you are using in UNIX/Linux – type `echo $SHELL`. Now SHELL is a variable name and its uppercase. So $SHELL will report back to Shell that we are currently using and it reports /bin/bash where bash is Bourne-again SHELL.

`echo $SHELL` – gives you the type of the shell

`echo $HOME` – happens to be the parent directory of all user specific directories

We use `nano` editor to create shell scripts with the SHELL. We usually save the shell scripts using `.sh` extension as it is a commonly used convention. `.sh` is for UNIX/Linux to understand that this is a script and does not hold a similar prominence for file extension as in regard to Windows OS. Include `#!/bin/bash` on top of the script to know that the script was written in BASH Shell. Below is the Hello World bash script – `script1.sh`. Line starting with `#` is a comment line.

```
#!/bin/bash
#Display a message
clear
echo "Hello World"
```

`Ctrl+O` to SAVE. `Ctrl+O` will write it out to the file system. So, the file name is already there. I'm going to press Enter and then I'll press `Ctrl+X` to exit. To execute this script `./script1.sh` needs to be called from the Shell prompt. To execute a file, we need to have required permissions on the file. `ls -l` command will give us the permission matrix on the files available on the respective directory. `ls -a` will list all files along with the hidden files which are set with (.) in the beginning of their name.

Permissions set `-rw-r- -r- -` [*Owner*][*Owning Group*][*EveryoneElse*]. The permission set, we've got read and write for the owner. We've got read for the owning group. So as the owner is root. The owning group is also called root and then we've got read for everyone else. What we don't have is execute for anybody. So, for example, if I type in change mode or `chmod +x` to add the execute permission, and then we apply that to script1.sh and now let's do a `ls -l`, the permission set will be updated to `-rwxrwxrwx`. Notice that we've got an x now for the owner, for the owning group, and for everybody.

2. Environment Variables

An Environment variable is a dynamically named value that can affect the way of running processes on a computer. They are part of the environment in which a process runs.

`echo $PATH` – to print the path value of all the system defined environment variables

`env` – to print all the environment variables available in the system

3. Input-Output Variables

A fundamental aspect of working with Shell scripts is working with input, output, and returning those results. Below is the script for `script2.sh` written in `vi` editor. There are different other editors like `nano, vim, gedit` etc. When we execute this script as `./script2.sh`, we get to enter `project code` and we get the value echo on screen.

```
#!/bin/bash
clear
echo
#Continue after reading 4 chars
read -n 4 -p "Enter Project Code:" project_code
echo
echo "Retrieving data for Project"  $project_code
echo
```

Variables are case sensitive in BASH. To unset a value associated with the variable, use `UNSET` keyword. By default, all the declared variables with values will be un-set as soon as we close the BASH automatically. However, if we wish to set the value permanently made available every time you login to BASH, go to `.bashrc` file to declare the variable and set value to it. This file will be called as part of the startup script. Few examples for variables

`pwd` – Current/Previous working directory

`echo "Current Usr:" $USER` – to print the user logged in

```
declare -i intvar
intvar=345
echo $intvar –
```
A way to declare and call the integer type variables.

```
declare -l rovar="Hyderabad"
echo $rovar
```
– A way to declare and call the string variables

`echo ${fakevar – "This is a test"}` – If returns blank then fakevar doesn't exists

`echo ${#rovar}` – gives the count of elements in the value of rovar `echo ${rovar#*r}` – truncates data until r and displays the rest of the value

4. Passing and Using Arguments in Shell Scripts

Below script is args.sh which defines $1 and $2 variables. Values need to be passed while executing the script as follows ./args.sh 31 27

```
#!/bin/bash
echo
if  test  "$1" = ""
        then
                echo "No first value supplied"
                exit
fi
if  test  "$2" = ""
        then
                echo "No second value supplied"
                exit
fi
clear
echo
echo "Sum of  values:" $1+$2='$1'+'$2'
```

5. Using Input Output Redirection

In BASH, > symbol prints data as output from file and < symbol takes data from the file as input. Pipe `|` symbol is used to provide response of one command to another. `tee` will help us to move the content of one command to another command.

Examples for redirection.

`touch labs.txt` – for creating a blank file. Using nano editor add data into to it

`cat labs.txt > /dev/stdout` – to print the data as output on Standard Output i.e. Screen

`sort < labs.txt` – to print the data in labs.txt in sorted fashion

`grep -i “t” < labs.txt` – finds all the strings/elements which has “t”

`find /<foldername>/  -size +800c` – finds files in <foldername> where there are more than 800 characters

`ls | wc -l` – takes the output of first command, passes it to second command and returns result.

`wc -l names.txt | tee -a file2.txt` – file2.txt contains the output of first command

6. Controlling and Manipulating the Shell Script

In BASH, we can move a process/script execution to background or foreground based on the time consumed by the script to run on open SHELL. There could a long running scripts which might take too much time. Such scripts can be moved to background. To demonstrate this, go to root folder and run `find` command. This will list all directories available on your root. Ideally, we use Cntrl+S to pause, Cntrl+Q to resume and Cntrl+ C to exit from a long running script.

Type `jobs` to know the list of active scripts running at a given point of time.

`cat /dev/random > /dev/null` – is a random number with no response

`cat /dev/random > /dev/null&` will run the editor and type text and stop

`fg <jobnumber>` - to resume the background running job on to screen

`bg <jobnumber>` - to move the job to background out of screen

`kill <jobnumber>` - to stop the job

7. Working with Built-in Variables

The command env gives us all the builtin variables used by the BASH Shell. HOSTTYPE, LANG, NAME, USER, SHELL, LOGNAME, PATH, LS_COLORS etc. are few of the built-in variables.

`ls /bin` - gives us all the files under `/bin` folder location.

`echo $?` – to get exit status of last executed command. If it returns `0`, then the last executed command was successful. Any non-zero value is bad news.

`ls /fake` – will throw error whenever there is no fake folder. Now `echo $?` Will return `1` which is error as the last executed threw error.

Example file to use built-in variables in a script - `machine_stats.sh`

```
#!/bin/bash
clear
echo "Computer Name:" $HOSTNAME
echo "Currently Logged in user:" $USER
echo "Number of this script:" $0
echo "Number of parameters passed to this script:" $#
echo
```

The script will return `0` as output value when executed as `./machine_stats.sh` as it was expecting an argument. Now pass `./machine_stats.sh 1`, we get output value as `1` as we passed as value along the file during its execution.

8. Using Manipulating Variables

While executing the below example script file output.sh, we do see that the values of variables are being manipulated and stored in an external file. The command `cat <hostname>_stats.txt` will give us the intended output from the script. Use `rm <hostname>_stats.txt` to delete the file.

```
clear
echo
read -p "Enter the City Name:" cityname
echo "Computer:" $HOSTNAME >> $HOSTNAME"_stats.txt"
echo "City:" $cityname
echo "Linux Kernel info:" `uname -a` >> $HOSTNAME"_stats.txt"
echo "Shell version:" $BASH_VERSION >> $HOSTNAME"_stats.txt"
clear
echo
echo $HOSTNAME"_stats.txt file written successfully."
echo
```

A variable has a scope or limit of access based on where it can be operated. While executing the below script `call_export_var.sh` file, we used `servername` variable from this file and called it in a subfile called `export_var.sh` which was declared in the same script. By calling `./call_export_var.sh`, we do not see the variable name as output. However, if we call `./export_var.sh` script file, we do see the variable name being displayed from `call_export_var.sh` file. The variables have been manipulated in such a way that they can be declared in one file and can be called in another script as this defined the scope. To do so we need to use a command called `export` within the parent script so that this exported variable can be used any other child script.

```
call_export_var.sh

servername="Prod1"
export servername
./export_var.sh

export_var.sh
clear
echo
echo "The Servername  is"$servername
echo
```

To understand scope better, follow below `scope.sh`. The variables in the functions are localized until they are called outside. If `func2` was called before `func1` declaration, it returns not results. Thus, `func2` should be called after `func2` logic. Same with the case of `func1`.

```
func1()
{
        #declare sets the variable scope to local within this function
        #declare  classname="BTech First Year"
        classname="BTech First Year"
}

func2()
{
        func1
        echo $classname
}
func2
```

By calling the scope.sh script file as . ./scope.sh is called ***dotsouring***, we can use the $classname variable outside the scope.sh file as dotsouring helps us re-use it in multiple places across the bash. Here the variable is declared internally in a script file and the output value is being used across the shell. `declare` command need to be used to ensure that it is used in such a way.

9. Formatting Output Variables

BASH allows output variable formatting. Follow below examples:

`echo "Kevin said "Hello World""` - we get Kevin said Hello World

`echo "Kevin said \"Hello World\""` - we get Kevin said "Hello World"

`echo "Stock Price is $500"` - we get Stock Price is 500

`echo "Stock Price is \$500"` - we get Stock Price is $500

`date` - gives us Thu Jan 3 22:18:42 DST 2019

`date +'%Y-%m-%d` - 2018-04-09

```
datevar= `date +'%Y-%m-%d`
```
`echar $datevar` - gives us 2018-04-09. ` *backtick* – is used to store results for the command in the variable

`printf "%s\n" $costcenter` - %s for string and \n is the newline character

`printf "%.3s\n" $constcenter .3s\n` - displays only first 3 strings

```
numvar=5.5
```

`echo $numvar` - gives 5.500000 with 6-digit precision

`printf "%f\n" $numvar` - `%f` for floating point number

```
numvar=5673,
printf "%d\n" $numvar
```
- `%d` for integer

10. Creating Functions

Function is a code snippet which is defined and used to re-run or re-use a logic. Below script `func_lib.sh` contains two functions `userinfo()` and `ipinfo()` which are called while running the script file as `./func_lib.sh`. By ***dotsouring*** `. ./func_lib.sh`, we can make `ipinfo` function run outside the script as an individual function as we have raised the scope of this function from local to global.

```
function userinfo()
{
        echo "Current username:" $USER
        echo "User Home directory path:" $HOME
}

function ipinfo()
{
        ip_var=`hostname -I | awk '{ print $1}'`
        echo "IP ADDRESS:" $ip_var
}

userinfo
ipinfo
```