

Linux Shell Scripts

What is Shell Script ?

- We have seen some basic shell commands , it's time to move on to scripts.
- There are two ways of writing shell programs.
 - You can type a sequence of commands and allow the shell to execute them interactively.
 - You can store those commands in a file that you can then invoke as a program. This is known as Shell Script.
- We will use bash shell assuming that the shell has been installed as ***/bin/sh*** and that it is the default shell for your login.

Why Shell Script ?

- Shell script can take input from user, file and output them on screen.
- Useful to create own commands.
- Save lots of time.
- To automate some task of day today life.
- System administration part can be also automated.

How to write and execute ?

- Use any editor to write shell script.
- The extension is *.sh*.
- After writing shell script set execute permission for your script.
 - *chmod +x script_name*
- Execute your script
 - *./script_name*

Shell script format

- Every script starts with the line
 - ***#!/bin/bash***
- This indicates that the script should be run in the bash shell regardless of which interactive shell the user has chosen.
- This is very important, since the syntax of different shells can vary greatly.
- *#* is used as the comment character.
- A word beginning with *#* causes that word and all remaining characters on that line to be ignored.

A sample shell script

```
#!/bin/bash  
echo "Hello User"  
echo "See the files in current directory"  
ls
```

Variables

- In Linux (Shell), there are two types of variable:
 - System variables - created and maintained by Linux itself.
 - ***echo \$USER***
 - ***echo \$PATH***
 - User defined variables - created and maintained by user.
- All variables are considered and stored as strings, even when they are assigned numeric values.
- Variables are case sensitive.

Variables

- When assigning a value to a variable, just use the name.
- No spaces on either side of the equals sign.
 - ***var_name=value***
- Within the shell we can access the contents of a variable by preceding its name with a **\$**.

myname=A [use quotes if the value contains spaces]

myos=Linux

text = 1+2

echo Your name:\$myname [A]

echo Your os:\$myos [Linux]

echo \$text [1+2]

Variables

- If you enclose a \$variable expression in double quotes, it's replaced with its value when the line is executed.
- If you enclose it in single quotes, no substitution takes place. You can also remove the special meaning of the \$ symbol by prefacing it with a \.

```
myvar="Hello"
```

```
echo $myvar [ Hello ]
```

```
echo "$myvar" [ Hello ]
```

```
echo '$myvar' [ $myvar ]
```

```
echo \ $myvar [ $myvar ]
```

Read

- To read user input from keyboard and store it into a variable use ***read var1,var2,.....varn***

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

```
echo -n "Enter your name:"
```

```
read name
```

```
echo -n "Enter your student no:"
```

```
read stdno
```

```
echo "Your Name:$name"
```

```
echo "Your Age:$stdno"
```

Shell Arithmetic

- The *expr* command evaluates its arguments as an expression.
- It is commonly used for simple arithmetic operations.

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

```
expr 1 + 1
```

```
expr 1 - 1
```

```
expr 1 \* 1
```

```
expr 1 / 1
```

```
var=`expr 1 + 1`
```

```
x=1
```

```
x=`expr $x + 1`
```

Shell Arithmetic

Expression Evaluation	Description
<code>expr1 expr2</code>	<code>expr1</code> if <code>expr1</code> is nonzero, otherwise <code>expr2</code>
<code>expr1 & expr2</code>	Zero if either expression is zero, otherwise <code>expr1</code>
<code>expr1 = expr2</code>	Equal
<code>expr1 > expr2</code>	Greater than
<code>expr1 >= expr2</code>	Greater than or equal to
<code>expr1 < expr2</code>	Less than
<code>expr1 <= expr2</code>	Less than or equal to
<code>expr1 != expr2</code>	Not equal
<code>expr1 + expr2</code>	Addition
<code>expr1 - expr2</code>	Subtraction
<code>expr1 * expr2</code>	Multiplication
<code>expr1 / expr2</code>	Integer division
<code>expr1 % expr2</code>	Integer modulo

If-Else

```
if [ condition1 ]; then  
    statement1  
elif [ condition2 ]; then  
    statement2  
else  
    statement3  
fi
```

- It is must to put spaces between the [braces and the condition being checked.
- If you prefer putting then on the same line as *if*, you must add a semicolon to separate the test from the *then*.

If-Else

String Comparison	Result
<code>string1 = string2</code>	True if the strings are equal.
<code>string1 != string2</code>	True if the strings are not equal.
<code>-n string</code>	True if the string is not null.
<code>-z string</code>	True if the string is null (an empty string).

Arithmetic Comparison	Result
<code>expression1 -eq expression2</code>	True if the expressions are equal.
<code>expression1 -ne expression2</code>	True if the expressions are not equal.
<code>expression1 -gt expression2</code>	True if <code>expression1</code> is greater than <code>expression2</code> .
<code>expression1 -ge expression2</code>	True if <code>expression1</code> is greater than or equal to <code>expression2</code> .
<code>expression1 -lt expression2</code>	True if <code>expression1</code> is less than <code>expression2</code> .
<code>expression1 -le expression2</code>	True if <code>expression1</code> is less than or equal to <code>expression2</code> .
<code>! expression</code>	True if the expression is false, and vice versa.

If-Else

File Conditional	Result
<code>-d file</code>	True if the file is a directory.
<code>-e file</code>	True if the file exists. Note that, historically, the <code>-e</code> option has not been portable, so <code>-f</code> is usually used.
<code>-f file</code>	True if the file is a regular file.
<code>-g file</code>	True if set-group-id is set on <code>file</code> .
<code>-r file</code>	True if the file is readable.
<code>-s file</code>	True if the file has nonzero size.
<code>-u file</code>	True if set-user-id is set on <code>file</code> .
<code>-w file</code>	True if the file is writable.
<code>-x file</code>	True if the file is executable.

If-Else

```
#!/bin/bash  
echo "Enter first number "  
read num1  
echo "Enter second number"  
read num2  
if [ $num1 -gt $num2 ]; then  
    echo "$num1 is greater than $num2"  
elif [ $num1 -lt $num2 ]; then  
    echo "$num1 is less than $num2"  
else  
    echo "$num1 and $num2 are equal"  
fi
```


Case

```
case $var in  
    condition1) statement ;;  
    condition2) statement ;;  
    *) statement3  
esac
```

- Notice that each pattern line is terminated with double semicolons *;;*.
- You can put multiple statements between each pattern and the next, so a double semicolon is needed to mark where one statement ends and the next pattern begins.

Case

```
#!/bin/sh  
echo "Is it morning? Please answer yes or no"  
read timeofday  
case "$timeofday" in  
    yes) echo "Good Morning";;  
    no ) echo "Good Afternoon";;  
    y )  echo "Good Morning";;  
    n )  echo "Good Afternoon";;  
    * )  echo "Sorry, answer not recognized";;  
esac
```

Case

```
#!/bin/sh  
echo "Is it morning? Please answer yes or no"  
read timeofday  
case "$timeofday" in  
    yes | y | Yes | YES ) echo "Good Morning";;  
    n* | N* )             echo "Good Afternoon";;  
    * )                   echo "Sorry, answer not recognized";;  
esac
```

Command Line arguments

- Command line arguments can be passed to the shell scripts. There exists a number of built in variables
 - **`$*`** - command line arguments
 - **`$#`** - number of arguments
 - **`$n`** - nth argument in `$*`
- ***`./script_name arg1 arg2 argn`***

For

```
for variable in list  
do  
    statement  
done
```

```
for (( expr1; expr2; expr3 ))  
do  
    statement  
done
```

For

[1]

#!/bin/bash

***echo "the number of args is
\$#"***

a=1

***for i in \$*
do***

echo "The \$a No arg is \$i"

a=`expr \$a + 1`

done

[2]

#!/bin/bash

***for i in `ls`
do***

echo \$i

done

[3]

***for((i=0;i<=50;i++))
do***

echo \$i

done

While

*while condition
do statements
done*

```
#!/bin/bash  
password="abc"  
echo "Enter password"  
read pass  
while [ $pass != $password ]  
do  
    echo "Wrong Password, Try again"  
    read pass  
done  
echo "Right Password"
```

Until

<i>until condition do</i>	<i>#!/bin/bash</i>
<i>statements</i>	<i>password="abc"</i>
<i>done</i>	<i>echo "Enter password"</i>
	<i>read pass</i>
	<i>until [\$pass = \$password]</i>
	<i>do</i>
	<i>echo "Wrong Password, Try again"</i>
	<i>read pass</i>
	<i>done</i>
	<i>echo "Right Password"</i>

Functions

- Functions can be defined in the shell and it is very useful to structure the code.
- To define a shell function simply write its name followed by empty parentheses and enclose the statements in braces.

```
function_name () {  
    statements  
}
```

- Function must be defined before one can invoke it.

Functions

```
#!/bin/sh  
foo() {  
    echo "Function foo is executing"  
}  
echo "script starting"  
foo  
echo "script ending"
```

output
script starting
Function foo is executing
script ending

Functions

- When a function is invoked, the parameters to the script *[**\$***, **\$#**, **\$1**, **\$2**]* and so on are replaced by the parameters to the function.
- When the function finishes, they are restored to their previous values.

```
#!/bin/bash  
showarg()  
{  
    a=1  
    for i in $*  
    do  
        echo "The $a No arg is $i"  
        a=`expr $a + 1`  
    done  
}  
echo "Listing start"  
showarg $*  
echo "Total:$#"  
echo "Listing End"
```

Functions

- Functions can return numeric values using the return command, use \$?

```
f() { return 10; }
```

```
f
```

```
result=$?
```

- Functions can also return strings by the following ways

```
[1]
```

```
f(){ var="123"; }
```

```
f
```

```
echo $var
```

```
[2]
```

```
f(){ echo "123"; }
```

```
result="$(f)"
```

Functions

```
#!/bin/sh
yes_or_no()
{
    echo "Is your name $* ?"
    echo "Enter yes or no:"
    read x
    case "$x" in
        y|yes) return 0;;
        n|no) return 1;;
    esac
}
```

```
if yes_or_no "$1"
then
    echo "Hi $1, nice name"
else
    echo "Never mind"
fi
```

Functions

- Be careful :
 - Function calling can be recursive.

```
f()  
{  
    statements  
f  
}  
f
```

- The parameter must be passed every time a function is invoked either from main or from any other functions.

Thanks