UNIX Shell Scripting

Preface

This document will take you through UNIX Shell Scripting.

Table of Contents

1. Shell Basics

The shell provides you with an interface to the UNIX system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

1.1 Shell Prompt

The prompt, $, which is called command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

The shell reads your input after you press Enter. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of **date** command which displays current date and time:

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using environment variable PS1 explained in Environment tutorial.

1.2 Shell Types

In UNIX there are two major types of shells:

1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the # character.

The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow:

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.

The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".

The Bourne shell is usually installed as /bin/sh on most versions of UNIX. For this reason, it is the shell of choice for writing scripts to use on several different versions of UNIX.
In this tutorial, we are going to cover most of the Shell concepts based on Borne Shell.

1.3 Shell Scripts

A shell script is a script written for the shell, or command line interpreter, of an operating system. Typical operations performed by shell scripts include file manipulation, program execution, and printing text.

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by a pound sign, #, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell scripts and functions are both interpreted. This means they are not compiled.

We are going to write a many scripts in the next several tutorials. This would be a simple text file in which we would put our all the commands and several other required constructs that tell the shell environment what to do and when to do it.

2. Writing 1st Shell Program

To ensure that the correct shell is used to run the script, you must add the following "magic" line to the beginning of the script:

```
#!/bin/bash
$ cd /usr/local/bin
```

2.1 The Magic of #!/bin/bash

The # !/bin/bash must be the first line of a shell script in order to run the script. If this appears on any other line, it is treated as a comment and ignored by all shells.

2.2 Comments

In shell scripts, comments start with the # character. Everything between the # and end of the line are considered part of the comment and are ignored by the shell.

```
        #!/bin/bash
# This is a simple script

echo "Hello World"
```

```
$ cd /usr/local/bin
```

2.3 echo Command

The echo command is mostly used for printing strings.

2.4 Calling A Script

source filename – Needs no execute permission
sh filename – Needs no execute permission

. filename – Needs no execute permission
filename – Needs execute permission

3. Variables

3.1 SHELL VARIABLES

The shell provides with the capability to define variables and assign them values . A variable name is a sequence of characters beginning a letter or an underscore.

There is no type associated with a shell variable. Every value that is assigned to a variable is treated as a string of characters by the shell.

Syntax :

```
variable_name=value

fruit=apple
name="satish mongam"
num1=23
fruit_name=banana
```

NOTE : There must be no spaces arround the " = " sign.

In order to use spaces you need to quote the value.

$ FRUIT="apple orange plum"
$ FRUIT='apple orange plum'

3.2 SPECIAL VARIABLES

Variable	Function
?	The previous command's exit status.
$	The PID of the current shell process.
!	The PID of the last command that was run in the background.
0	The filename of the current script.
1-9	The first through ninth command-line arguments given when the current script was invoked: $1 is the value of the first command-line argument, $2 the value of the second, and so forth.
_	The last argument given to the most recently invoked command before this one.

3.3 ACCESSING VALUES

To access the value stored in a variable, prefix its name with the dollar sign ($).

echo $FRUIT

3.4 ARRAY VARIABLES

Arrays provide a method of grouping a set of variables. There are 3 methods to define an array variable.

3.4.1 Method One

name[index]=value

```
Example:
$ FRUIT[0]=apple
$ FRUIT[1]=banana
$ FRUIT[2]=orange
```

3.4.2 Method Two

name=(value1 ... valuen)

Example :

```
$ band=(derri terry mike gene)

is equivalent to the following commands:
$ band[0]=derri
$ band[1]=terry
$ band[2]=mike
$ band[3]=gene
```

3.4.3 Method Three

myarray=([0]=derri [3]=gene [2]=mike [1]=terry)

3.5 ACCESSING ARRAY VALUES

After you have set any array variable, you access it as follows:

Syntax :

```
{name[index]}
```

```
Eg : echo ${FRUIT[2]}
```

You can access all the items in an array in one of the following ways:
${name[*]}

```
Eg : $ echo ${FRUIT[*]}

output:
apple banana orange
```

3.6 Unsetting Variables

```
unset name

export FRUIT=APPLE;
echo $FRUIT
output : apple

unset FRUIT
```

4. Operators

There are various operators supported by each shell. Our tutorial is based on default shell (Bourne) so we are going to cover all the important Bourne Shell operators in the tutorial.

There are following operators which we are going to discuss:

- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

Here is simple example to add two numbers:

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

This would produce following result:

```
Total value : 4
```

There are following points to note down:

- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expression should be enclosed between ``, called inverted commas.

4.1 Arithmetic Operators

There are following arithmetic operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then:

Show Examples

Description	Example
Addition - Adds values on either side of the operator	`expr $a + $b` will give 30
Subtraction - Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
Multiplication - Multiplies values on either side of the operator	`expr $a \* $b` will give 200
Division - Divides left hand operand by right hand operand	`expr $b / $a` will give 2
Modulus - Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
Assignment - Assign right operand in left operand	a=$b would assign value of b into a
Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
Not Equality - Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

4.2 Relational Operators

Bourne Shell supports following relational operators which are specific to numeric values. These operators would not work for string values unless their value is numeric.

For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable a holds 10 and variable b holds 20 then:

Show Examples

Description	Example
Checks if the value of two operands are equal or not, if yes then condition becomes true.	`[ $a -eq $b ] is not true.`
Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	`[ $a -ne $b ] is true.`
Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	`[ $a -gt $b ] is not true.`
Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	`[ $a -lt $b ] is true.`
Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	`[ $a -ge $b ] is not true.`
Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	`[ $a -le $b ] is true.`

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a <= $b] is correct where as [$a <= $b] is incorrect.

4.3 Boolean Operators

There are following boolean operators supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then:

Show Examples

Operator	Description	Exampl
!	This is logical negation. This inverts a true condition into false and vice versa.	`[ ! false ] is true`
-o	This is logical OR. If one of the operands is true then condition would be true.	`[ $a -lt 20 -o $b -gt 100 ] is true.`
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.	`[ $a -lt 20 -a $b -gt 100 ] is false.`

4.4 String Operators

There are following string operators supported by Bourne Shell.

Assume variable a holds "abc" and variable b holds "efg" then:

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	`[ $a = $b ] is not true.`
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	`[ $a != $b ] is true.`
-z	Checks if the given string operand size is zero. If it is zero length then it returns true.	`[ -z $a ] is not true.`
-n	Checks if the given string operand size is non-zero. If it is non-zero length then it returns true.	`[ -z $a ] is not false.`
str	Check if str is not the empty string. If it is empty then it returns false.	`[ $a ] is not false.`

4.5 File Test Operators:

There are following operators to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

Show Examples

Operator	Description	Example
-b file	Checks if file is a block special file if yes then condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file if yes then condition becomes true.	[-b $file] is false.
-d file	Check if file is a directory if yes then condition becomes true.	[-d $file] is not true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe if yes then condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.	[-t $file] is false.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute if yes then condition becomes true.	[-x $file] is true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.	[-s $file] is true.
-e file	Check if file exists. Is true even if file is a directory but exists	[-e $file] is true.

5. SHELL INPUT WITH READ COMMAND

When writing shell programs ,you might need to enter some data from key board and assign it to the shell variables. The shell statement read solves this purpose.

Syntax :

read var_name

Read command waits for the user to enter a value. The shell assigns that value to the shell variable specified by var_name.

```
2)
# !/bin/bash
# Reads input from user

echo "Enter your name  : "
read name

echo "Hai $name Welcome to Miracle World "
```

6. Shell Substitutions

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example:
Following is the example, while printing value of the variable its substitued by its value. Same time "\n" is substituted by a new line:

```
#!/bin/sh

a=10
echo -e "Value of a is $a \n"
```

This would produce following result. Here **-e** option enables interpretation of backslash escapes.
Value of a is 10

Here is the result without -e option:

```
Value of a is 10\n
```

Here are following escape sequences which can be used in echo command:

Escape	Description
\\	backslash
\a	alert (BEL)
\b	backspace
\c	suppress trailing newline
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab

You can use **-E** option to disable interpretation of backslash escapes (default).

You can use **-n** option to disable insertion of new line.

6.1 Command Substitution:

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Syntax:

The command substitution is performed when a command is given as:

```
`command`
```

When performing command substitution make sure that you are using the backquote, not the single quote character.

Example:

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrate command substitution:

```
#!/bin/sh

DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

This will produce following result:

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

7. QUOTING

Turning off the special meaning of a character is called quoting, and it can be done three ways:

- Using the backslash (\)
- Using the single quote (')
- Using the double quote (")

7.1 Quoting with Backslashes :

Putting a backslash (\)in front of the a character to take away its special meaning, enabling you to display it as a literal character.

Eg: *name=satish*

```
echo $name
satish

echo \$name
$name
```

7.2 Using Double Quotes :

Double quotes take away the special meaning of all characters except the
1) ($) for parameter substitution
2) (`) Backquotes for command substitution
3) (\) Backslashes.

Eg:

```
name=satish
echo "My name is $name"
echo "My name is \$name"
```

My name is satish
My name is $name

7.3 Using Single Quotes :

Any characters within single quotes are quoted just as if a backslash is in front of each character. All characters inside single quotes are interepted with no special meaning .

Eg :

```
name=satish
surname=mongam

echo "$name $surname"
echo '\$name \$surname'

satish mongam
\$name \$surname
```

8. IO Redirections

Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from a place called standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is also your terminal by default.

8.1 Output Redirection:

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal:

Check following **who** command which would redirect complete output of the command in users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check *users* file then it would have complete content:

```
$ cat users
oko          tty01    Sep 12 07:30
ai           tty15    Sep 12 13:32
ruth         tty21    Sep 12 10:10
pat          tty24    Sep 12 13:07
steve        tty25    Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example:

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows:

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

8.2 Input Redirection:

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows:

```
$ wc -l users
2 users
$
```

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file *users*:

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

8.3 Here Document:

A *here document* is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is:

```
command << delimiter
document
delimiter
```

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the here document has completed. Without it, the shell continues to read input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command **wc -l** to count total number of line:

```
$wc -l << EOF
            This is a simple lookup program
            for good (and bad) restaurants
            in Cape Town.
EOF
3
$
```

You can use *here document* to print multiple lines using your script as follows:

```
#!/bin/sh

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

This would produce following result:

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

The following script runs a session with the vi text editor and save the input in the file test.txt.

```
#!/bin/sh

filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

If you run this script with vim acting as vi, then you will likely see output like the following:

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file test.txt:

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

8.4 Discard the output:

Sometimes you will need to execute a command, but you don't want the output displayed to the screen. In such cases you can discard the output by redirecting it to the file /dev/null:

```
$ command > /dev/null
```

Here command is the name of the command you want to execute. The file /dev/null is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect STDERR to STDOUT:

```
$ command > /dev/null 2>&1
```

Here 2 represents STDERR and 1 represents STDOUT. You can display a message on to STDERR by redirecting STDIN into STDERR as follows

9. The expr Utility

We use ***expr*** utility to perform arithmetic operations.

```
# !/bin/bash
# Adding 2 numbers

num1=23
num2=30
total=`expr $num1 \+ $num2`
```

10. FLOW CONTROL

Two powerful flow control mechanics are available in the shell:
1) The if statement 2) The case statement

10.1 IF Statement

Syntax:

```
if list1
then
            list2
elif list3
then
            list4
else
            list5
fi
```

Example :

```
# !/bin/bash

echo -n "Enter Two Numbers : "

read num1 num2

if [ $num1 -gt $num2 ] ; then
```

```
            echo "First Number Is Big"
else
            echo "Second Number Is Big"
fi
```

10.2 Case Statement

The basic syntax is

```
case word in
            pattern1)
             list1
             ;;
            pattern2)
             list2
             ;;
esac

OR

case word in
            pattern1) list1 ;;
            pattern2) list2 ;;
esac
```

Here the string word is compared against every pattern until a match is found. The list following the matching pattern executes. If no matches are found, the case statement exits without performing any action. There is no maximum number of patterns, but the minimum is one.

Example :

```
# !/bin/bash

echo "Enter A Word : "
read word

case "$word" in
           apple) echo "Apple pie is quite tasty." ;;
           banana) echo "I like banana nut bread." ;;
           kiwi) echo "New Zealand is famous for kiwi." ;;
           *) echo "No Matching Fruit Try Again";;
esac
```

11. Loops

Two main types of loops are
1) The while loop 2) The for loop

11.1 while Loop

The while loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax :

```
while command
do
           list
done
```

Example :

```
# !/bin/bash
x=0
while [ $x -lt 5 ]
do
           echo $x
           x=´echo "$x + 1"´
done
```

Its output looks like this:
0
1
2
3
4
5

11.2 The until Loop :

Syntax :

```
until command
do
            list
done
```

Example :

```
# !/bin/bash

x=1
while [ ! $x -ge 5 ]
do
            echo $x
            x=´echo "$x + 1"´
done
```

is equivalent to the following until loop:

```
# !/bin/bash

x=1;
until [ $x -ge 5 ]
do
            echo $x
            x=´echo "$x + 1"´
done
```

12.2) The for and select Loops :

Both the for and select loops operate on lists of items. The for loop repeats a set of commands for every item in a list, whereas the select loop enables the user to select an item from a list.

11.3 for Loop :

```
for name in word1 word2 ... wordN
do
list
done
```

Here name is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable name is set to the next word in the list of words, word1 to wordN. The first time, name is set to word1; the second time, it's set to word2; and so on.

```
# !/bin/bash

for i in 0 1 2 3 4 5
do
echo $i
done
```

This loop counts to nine as follows:
0

1
2
3
4
5

11.4 The select Loop :

The select loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

Syntax :

```
select name in word1 word2 ... wordN
do
          list
done
```

Here name is the name of a variable and word1 to wordN are sequences of characters separated by
spaces (words). The set of commands to execute after the user has made a selection is specified by list.

```
# !/bin/bash

select COMPONENT in num1 num2 num3 all quit
do
case $COMPONENT in
          comp1|comp2|comp3) echo "You are selected $COMPONENT" ;;
          all) echo "comp1 comp2 comp3";;
          quit) break ;;
          *) echo "ERROR: Invalid selection, $REPLY." ;;
esac
done
```

12 The test Command

The test command is used to evaluate conditions. You'll notice that the preceding examples include square brackets around the conditions to be evaluated. The square brackets are syntactically equal to the test command. For example, the preceding script could have been written:

```
if ( test $COLOR="purple" )
```

This concept is important because the test command has a number of options that can be used to evaluate all sorts of conditions, not just simple equality. For example, use test to see whether a particular file exists:

```
if ( test -e filename )
```

In this case, the test command would return a value of true, or 0, if the file exists, and false, or 1, if it doesn't.

You can get the same effect by using square brackets:

if [-e *filename*]

The following table shows other options you can use with test or with square brackets:

Option	Test Condition
-d	The specified file exists and is a directory.
-e	The specified file exists.
-f	The specified file exists and is a regular file (not a directory or other special file type).
-G	The file owner's group ID matches the file's ID.
-nt	The file is newer than another specified file (takes the syntax *file1* -nt *file2*).
-ot	The file is older than another specified file (takes the syntax *file1* -ot *file2*).
-O	The user issuing the command is the file's owner.
-r	The user issuing the command has read permission for the file.
-s	The specified file exists and is not empty.
-w	The user issuing the command has write permission for the file.
-x	The user issuing the command has execute permission for the file.