**Shell Programming**

The UNIX shell program interprets user commands, which are either directly entered by the user, or which can be read from a file called the shell script or shell program. Shell scripts are interpreted, not compiled. The shell reads commands from the script line by line and searches for those commands on the system.

**Shell script**

Shell script is a file-containing list of commands to be executed in a particular order.

A good shell script will have comments, (a statement preceded by a pound sign,**#)**, describing the purpose of the statement.

In a script we can use conditional tests, such as value A is greater than value B, loops or iterative statements to execute some steps repetitively or to navigate through a list of data or records and do processing. We can use files to read and store data. Can use variables to read and store data. A script may include functions also.

When a script is executed, the shell reads the commands one by one and executes them .

We can create the simple shell script file by using vi editor or cat command like,

$ vi test.sh

$ cat > test.sh

Below mentioned **shebang** statement should be the first statement in the shell script as it tells the system that the commands mentioned in the shell script are to be executed by the shell /bin/sh

**#!/bin/sh**

Consider the shell script with just two commands pwd & ls.

$cat test.sh

#!/bin/bash

pwd

ls

**Importance of shell script**

Shell scripts are basically used for automating processes that we repeat at the prompt .

Following are some activities we can do using shell scripts:

* Automation of repetitive task
* Creating our own power tools/utilities.
* Automating command input or entry.
* Customizing administrative tasks.
* Creating simple applications.
* Automating administration tasks such as adding new users, removing obsolete users etc

Some Practical examples where shell scripting can be actively used:

* Monitoring your Linux system.
* Data backup and creating snapshots.
* Dumping Oracle or MySQL database for backup.
* Creating email based alert system.
* Find out what processes are eating up your system resources.
* Find out available and free memory.
* Find out all logged in users and what they are doing.
* Find out if all necessary network services are running or not. For example if web server failed then send an alert to system administrator via a pager or an email.
* Find out all failed login attemp. If login attempt are continued repeatedly from same network IP, automatically block all those IPs accessing your network/service via firewall.
* User administration as per your own security policies.
* Find out information about local or remote servers.
* Configure server such as BIND (DNS server) to add zone entries.

**Executing a shell script**

Below two methods can be used to execute the shell script.

$ sh filename

Or

$ ./filename

In this case we have to modify the file access permissions of  the shell script before execution.

To provide execute permission , following command is used.

$ chmod u+x filename

**Basic Operators in shell scripting**

Below operators are supported by shell.

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

Here is simple example to add two numbers:

**Example:**

#!/bin/sh

val=`expr 2 + 2`

echo "Total value : $val"

**Output:**

$ 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 to execute expr command correctly.

**Arithmetic Operators**

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | `expr $a + $b ` will give 30 |
| - | Substraction | `expr $a - $b ` will give -10 |
| \* | Multiplication | `expr $a \\* $b` will give 200 |
| / | Division | `expr $b / $a` will give 2 |
| % | Modulus | `expr $a % $b` will give 0 |
| != | Not equal | [ $a != $b ] will give true |
| = | assignment | a=$b will assign value of b to a. |
| == | Equality | [ $a == $b ] will return false. |

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.

**Relational Operators**

Below are 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:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-eq** | Check if the values of 2 operands are equal or not, if yes then condition becomes true. | [ $a -eq $b ] is false |
| **-ne** | Check if the values of 2 operands are equal or not, if values are not equal then condition becomes true. | [ $a -eq $b ] is true |
| **-gt** | Check if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | [ $a -gt $b ] is false |
| **-lt** | Check 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 |
| **-ge** | Check 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 false |
| **-le** | Check 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.

**Boolean Operators**

Assume variable a holds 10 and variable b holds 20 then

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **!** | 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. |

**String Operators**

These are string operators. Assume variable a holds "abc" and variable b holds "efg" then:

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

**File Test Operators:**

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| **-b file** | Returns true, if file is a block special file | [ -b $file ] is false. |
| **-c file** | Returns true, if file is a character special file | [ -b $file ] is false. |
| **-d file** | Returns true, Check if file is a directory | [ -d $file ] is not true. |
| **-f file** | Returns true, Check if file is an ordinary file or special file | [ -f $file ] is true. |
| **-r file** | Returns true, Checks if file is readable | [ -r $file ] is true. |
| **-w file** | Returns true, Check if file is writable | [ -w $file ] is true. |
| **-x file** | Returns true, Check if file is execute | [ -x $file ] is true. |
| **-s file** | Returns true, Check if file has size greater than 0 | [ -s $file ] is true. |
| -**e file** | Returns true, Check if file exists | [ -e $file ] is true. |

**Wild Card Characters**

Symbol used to replace or represent one or more characters. Wildcards or wild characters are either asterisk (\*), which represent one or more characters or question mark (?), which represent a single character.

|  |  |  |  |
| --- | --- | --- | --- |
| **Wild card /Shorthand** | **Meaning** |  | **Examples** |
| **\*** | Matches any string or group of characters. | **$ ls \*** | will show all files |
| **$ ls a\*** | will show all files whose first name is starting with letter 'a' |
| **$ ls \*.c** | will show all files having extension .c |
| **$ ls ut\*.c** | Will show all files having extension .c but file name must begin with 'ut'. |
| **?** | Matches any single character. | **$ ls ?** | will show all files whose names are 1 character long |
| **$ ls fo?** | will show all files whose names are 3 character long and file name begin with fo |
| **[...]** | Matches any one of the enclosed characters | **$ ls [abc]\*** | Will show all files beginning with letters a,b,c |

**Note: [..-..]** A pair of characters separated by a minus sign denotes a range.

***Example*:**

**$ ls /bin/[a-c]\***

Will show all File name beginning with letter a,b or c.

***Output:***

/bin/arch /bin/awk /bin/bsh /bin/chmod /bin/cp /bin/as /bin/basename /bin/cat

**Shell Quoting Mechanism**

**The Metacharacters**

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

**Example:**

**?** Matches with a single character while listing files in a directory and an **\*** would match more than one characters.

Here is a list of most of the shell special characters (also called metacharacters):

\* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

**Example:**

#!/bin/sh echo Hello; Word

This would produce following result.

Hello ./test.sh: line 2: Word: command not found shell returned 127

Now let us try using a quoted character:

#!/bin/sh echo Hello\; Word

This would produce following result:

Hello; Word

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell:

#!/bin/sh echo "I have \$1200"

This would produce following result:

I have $1200

|  |  |
| --- | --- |
| **Quote** | **Description** |
| Single quote | All special characters between these quotes lose their special meaning. |
| Double quote | Most special characters between these quotes lose their special meaning with these exceptions:  $  `  \$  \'  \"  \\ |
| Backslash | Any character immediately following the backslash loses its special meaning. |
| Back Quote | Anything in between back quotes would be treated as a command and would be executed. |

**The Single Quotes**

Consider an echo command that contains many special shell characters:

echo <-$1500.\*\*>; (update?) [y|n]

Putting a backslash in front of each special character is tedious and makes the line difficult to read:

echo \<-\$1500.\\*\\*\>\; update\?update\?

y∥ny‖n

There is an easy way to quote a large group of characters. Put a single quote ( ') at the beginning and at the end of the string:

echo '<-$1500.\*\*>; (update?) [y|n]'

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you would precede that using a backslash (\) as follows:

echo 'It\'s Shell Programming

**The Double Quotes:**

Try to execute the following shell script. This shell script makes use of single quote:

VAR=ZARA

echo '$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]'

This would produce following result:

$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows:

VAR=ZARA

echo "$VAR owes <-\$1500.\*\*>; [ as of (`date +%m/%d`) ]"

Now this would produce following result:

ZARA owes <-$1500.\*\*>; [ as of (07/02) ]

Double quotes take away the special meaning of all characters except the following:

* $ for parameter substitution.
* Backquotes for command substitution.
* \$ to enable literal dollar signs.
* \` to enable literal backquotes.
* \" to enable embedded double quotes.
* \\ to enable embedded backslashes.
* All other \ characters are literal (not special).

**Back Quotes: Command Substitution**

Putting any Shell command in between back quotes would execute the command

**Syntax:**var=`command`

**Example:**

Following would execute date command and produced result would be stored in DATA variable.

DATE=`date` echo "Current Date: $DATE"

This would produce following result:

Current Date: Thu Jul

**Shell Decision Statement**

While writing a shell script, there may be situations when you need to adopt one path out of many available paths. In such cases you need to make use of conditional statements that allow your program to make correct decisions and perform right actions.

Unix Shell supports conditional statements, which are used to perform different actions based on different conditions.

Here we will explain following two decision-making statements

* The if...else statement
* The case...esac statement

**If..else statements**

We can use “if..else” statement which can be used as decision making statement to select an option from a given set of options.

Unix Shell supports following forms of if..else statement:

* if...fi statement
* if...else...fi statement
* if...elif...else...fi statement

**Syntax:**

* **if...fi statement**

if [condition]

then

     command(s)

fi

* **if...else...fi statement**

if [ condition(s) ] then

   command(s)

else

   command(s)

fi

* **if...elif...else...fi statement**

if [ condition(s) ]

then

     command(s)

elif [ condition(s) ] # Many elif-then allowed

then

   command(s)

else

   command(s)

fi

**test command in if condition**

We can use test command as condition of if condition as used in the below script.

$cat if\_test.sh

#!/bin/sh

echo “Do you want to quit (Y/y)? ”

read ans

if  test $ans ==’y’ –o $ans==’Y’

then

      exit

else

      echo “ to exit enter N or n”

fi

**case...esac Statement**

We can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

The interpreter checks each case against the value of the expression until it finds a match. If nothing matches, goes with the default condition.

**Syntax:**

|  |
| --- |
| **case** word **in**pattern1)  Statement(s) to be executed if pattern1 matches  ;;  pattern2)  Statement(s) to be executed if pattern2 matches  ;;  pattern3)  Statement(s) to be executed if pattern3 matches  ;;  **esac** |

Here string word is compared against every pattern until a match is found and the statement(s) following the match pattern executes. If shell cannot find any match, the case statement exits without performing any action. When statement(s) part is executed. The command **;;** indicates program flow should jump to the end of the entire case statement.

There is no maximum number of patterns, but the minimum is one.

**Example:**

#!/bin/sh

COURSE=”DB”

case “$COURSE” in

 “Java”) echo “Java is a programming language”

   ;;

 “Perl”)echo “Perl is scripting language”

  ;;

 “DB”)echo “Oracle is a DB”

  ;;

esac

**Output:**

Oracle is a DB

**Iterative Statements/Loop :**

Loops are a powerful programming tool that enables you to execute a set of commands repeatedly.

* while loop
* for loop
* until loop

**while loop**

Here condition is evaluated by shell at the beginning of each iteration. If the resulting value is true, given command(s) are executed. If condition is false then no command would be executed and program would jump to the next line after done statement.

**Syntax:**

while condition

do # executed as long as the condition is true

   command(s)

done

**Example:**

a=0

while [ $a -lt 3 ]

do

   echo $a

   a=`expr $a + 1`

done

**Output:**

0

1

2

**until loop**

Here condition is evaluated by shell. If the resulting value is false, given command(s) are executed. If condition is true then no command would be executed and program would jump to the next line after done statement.

**Syntax:**

until condition # complement of while

do

# executes the body as long as the condition is false

   command(s)

done

**Example:**

|  |  |
| --- | --- |
|  | a=0  until [ ! $a -lt 3 ] do  echo $a a=`expr $a + 1`  done |

**Output:**

0

1

2

**for loop**

For loop operate on lists of items. It repeats a set of commands for every item in a list.

**Syntax:**

for var in w1 w2 w3.. wn

do

      command #Statement(s) to be executed for every w in the list

done

**Example:**

for var in 0 1 2 3 4 5

do

     echo $var

done

**Output:**

0

1

2

3

4

5

**Example:**

|  |  |
| --- | --- |
|  | for filename in `ls tempdir`  do  echo “Displaying contents of ${filename}”  cat ${filename}  done |

We can use "break" and "continue" commands to control the loop.

Command "break" causes the termination of a loop and “continue” resumes execution at its top.

**String Handling**

* String handling with test command:

|  |  |  |  |
| --- | --- | --- | --- |
| test str | Returns true |  | if str is not null |
| test –n str | Returns true |  | if length of str is greater than zero |
| test –z str | Returns true |  | if length of str is equal to zero |

* String handling with expr command

The expr is quite handy for finding the length of a string and extracting a sub-string:

**Length of the string:**

|  |  |
| --- | --- |
|  | $ str=”abcdefghijk” ;  $ n=`expr "$str" : ‘.\*’` ;  $ echo $n  11 |

expr gave how many times any character (.\*) occurs. This feature is very useful in validating data entry.

**Extracting a sub-string:**

|  |  |
| --- | --- |
|  | $ str=”abcdefghijk” ;  $ expr “$str” : ‘……....’  gh |

Note that there are 6 dots preceding the sequence ..... This advanced regular expression signifies that the first six characters of the string are to be ignored and extraction should start from the 7th character. Two dots inside .... suggests that this extraction is limited to two characters only (backslashes override the usual interpretation of ‘()’).

**Extracting string from 3rd character to end of the string:**

|  |  |
| --- | --- |
|  | $ str="abcdefghijk"  $ expr "$str" : '...∗.∗'  cdefghijk |

**Location of first occurrence of a character “d” inside string:**

|  |  |
| --- | --- |
|  | $ str=”abcdefghijk” ;  $ expr "$str" : '[^d]\*d‘  4 |

**Location of last occurrence if a character inside string:**

Below will give the last occurrence of character 'a' from string str.

|  |  |
| --- | --- |
|  | $str=”abc def abc”  $expr "$str" : '[^u]\*a' |

1. **Command Line Arguments**

To make a shell script a generalized script, we should avoid hard coding. User should be able to provide the values required for processing as input while running the shell script. To facilitate this we have to use command line arguments.

The statement we write in the command prompt starting with the command followed by list of arguments is the command line argument.

Example:

$ ls dir1 dir2 dir3

A set of shell variables are there to represent the command line arguments.

$0 – represents the command name (first word on command line). For above example “ls”

$1 - the first argument of command (second word/symbol ).For the above example dir1

$2 – the second argument to the command. For the above example, dir2

In this way we can use up to $9

The command-line arguments $1, $2, $3...$9 are also called positional parameters, with $0 pointing to the actual command/program/shell script and $1, $2, $3, ...$9 as the arguments to the command.

Consider below sample shell script “test.sh”

$cat test.sh

|  |
| --- |
| #!/bin/sh  echo “Program name is $0” |
| echo “First argument is $1”  echo “Number of arguments passed =$#”  echo “The arguments are $@” |

Let’s execute the above shell script “test.sh”. Remember to change File access permission of script to execute

$ sh test.sh arg1 arg2 arg3

Program name is test

First argument is arg1

Second argument is arg2

Number of arguments passed = 3

The arguments are arg1 arg2 arg3

There’s another way to execute the script using **./**

$ ./test.sh arg1 arg2 arg3

Program name is test

First argument is arg1

Second argument is arg2

Number of arguments passed = 3

The arguments are arg1 arg2 arg3

**User Input: read**

To get input from the keyboard, you use the read command. The read command takes input from the keyboard and assigns it to a variable.

**Example: read.sh**

#!/bin/sh

echo –n “Enter some text > “

read text

echo “You entered: $text”

Note that “-n” given to the echo command causes it to keep the cursor on the same line; i.e., it does not output a carriage return at the end of the prompt.

Next, we invoke the read command with “text” as its argument. What this does is wait for the user ro type something followed by a carriage return (the Enter key) and then assign whatever was typed to the variable text.

**Execution:**

$sh read.sh

Enter some text > This is some text

You entered: This is some text

**​​​​​​​​​​​​​​Validation of Command Line Arguments**

Let us assume, we have one shell script which requires exactly 2 arguments to execute. Shell script should throw proper error message in case user has not given exactly two arguments.

Let us have a small shell script "test.sh" to show how to implement validations.

|  |
| --- |
| #!/bin/sh  if [ $# -ne 2 ]  then  echo " Not sufficient arguments, please give exact 2 arguments"  else  echo "Continue with our task" fi |

$ ./test.sh

Not sufficient arguments, please give exact 2 arguments

$ sh test.sh abc

Not sufficient arguments, please give exact 2 arguments

$ ./test.sh abc def ghi

Not sufficient arguments, please give exact 2 arguments

$ ./test.sh abc def 2

Not sufficient arguments, please give exact 2 arguments

$ ./test.sh abc def

Continue with our task

For scripts where input expected is a file , validations can be incorporated for file tests like whether the file given as input exists or not, whether it is empty or not, whether it is readable or not and so on as per requirements.