Unix Shell Scripting

Variables

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data. The shell enables you to create, assign, and delete variables.

Rules:

- Must begin with Letter or underscore (‘_’)
- Case sensitive
- No Special characters allowed in variable name except underscore (‘_’)

Types of Variable

SYSTEM Variables: Created by SHELL itself. These variables are useful to setting system environment. Eg : PATH, DISPLAY etc

User Defined Variables (UDV): Created and managed by user by user.

Local Variables − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.

Environment Variables − An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.

Creating a UDV: A UDV variable can be created simply by assigning a value (or NULL) to the variable or by using "declare" command.
Eg: NUM1=10, NAME="Sami"
declare NUM1=10, declare NUM2

Assigning Value: Variable can be assigned a value using "=" without spaces either side.
Eg : NUM1=10, NAME="Sami"

Accessing the value of Variable: Variable's value can be accessed by using '$' followed by variable name.
Eg: NAME2=$NAME, echo $NUM!

Delete Variable: A Variable can be deleted using "unset" command
Eg: unset NAME

Making Variable Readonly: Readonly variable's value cannot be changed or that variable cannot be deleted
Eg: readonly NAME

Declare options:

Declare statement used to create variables in shell scripts. The following options set the characteristics of variables.

-i	Integer type variable
-r	Readonly variable
-a	Array type of variable

Special Variables

Variable	Description
$0	The filename of the current script.
$<n>	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

echo command

display text or value of variable.

USAGE: echo [options] [string, variables...]

Options
-n Do not output the trailing new line.
-e Enable interpretation of the following backslash escaped characters in the strings:
\a alert (bell)
\b backspace
\c suppress trailing new line
\n new line
\r carriage return
\t horizontal tab
\\ backslash

Example : echo -e “Hello World \n Welcome”

read command

USAGE : read -p “Prompt Message” <Variable Name>

Allows user to assign the value to a variable by inputting from keyboard

Quoting

Quoting is used to accomplish two goals:

1. To control (i.e., limit) substitutions and
2. To perform grouping of words.

Strong Quotes: Single Quotes (“) . Enclosed text will be left alone with no variable or command substitution.

Eg: echo ‘Is your home directory $HOME?’

Weak Quotes: Double Quotes [“”]. Enclosed text will be expanded if it contains variables.

Back Ticks [` `]: Enclosed text assumed as command and will be executed

Examples:

A Script to ask name and print the given name

```
#!/bin/sh
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Shell Arithmetic

Arithmetic Operators

There are following arithmetic operators supported by Bourne Shell.

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder

Shell Arithmetic

In bash Shell Arithmetic is done using a external program ‘expr’ or ‘bc’ for floating point operations where as in Korn or C-Shell arithmetic can be without using ‘expr’ .

Eg:

```
N1=10
N2=20
echo   `expr   $N1 + $N2`
```

Note: Spaces around operator is mandatory.

Floating Point Arithmetic

Shell script are not meant for complex Arithmetic calculations but they support limited floating point operation using bc command.

Eg: `NUM1=`echo '1.2 + 2.2'|bc -l``

Comparision Operators

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

String Comparison Operators

Operator	Description
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes 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.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Boolean Operators

Operator	Description
!	This is logical negation. This inverts a true condition into false and vice versa.
-o	This is logical OR. If one of the operands is true then condition would be true.
-a	This is logical AND. If both the operands are true then condition would be true otherwise it would be false.

File Test Operators

Operator	Description
-d file	Check if file is a directory if yes then condition becomes true.
-f file	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.
-g file	Checks if file has its set group ID (SGID) bit set if yes then condition becomes true.
-k file	Checks if file has its sticky bit set if yes then condition becomes true.
-t file	Checks if file descriptor is open and associated with a terminal if yes then condition becomes true.
-u file	Checks if file has its set user id (SUID) bit set if yes then condition becomes true.
-r file	Checks if file is readable if yes then condition becomes true.
-w file	Check if file is writable if yes then condition becomes true.
-x file	Check if file is execute if yes then condition becomes true.
-s file	Check if file has size greater than 0 if yes then condition becomes true.
-e file	Check if file exists. Is true even if file is a directory but exists.