Shell scripting

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 # 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.

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

A **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.

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

Shell Types :-

In Unix, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

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

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct

#!/bin/sh

Example :-

```
#!/bin/sh


echo "What is your name?"
read  PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script −

```
$./test.sh
What is your name?
xyz
Hello, xyz
$
```

<u>Variables :-</u>

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names −

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

The reason you cannot use other characters such as **!**, *****, or **-** is that these characters have a special meaning for the shell.

<u>Variable Types :-</u>

When a shell is running, three main types of variables are present −

- **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 the command prompt.
- **Environment Variables** − An environment variable 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.

- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Defining Variables :-

Variables are defined as follows −

variable_name=variable_value

For example −

```
NAME="xyz"
```

Accessing Values :-

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

For example, the following script will access the value of defined variable NAME and print it

```
#!/bin/sh

NAME="xyz"
echo $NAME
```

control constructs :-

The if...else statements

If else statements are useful decision-making statements which can be used 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 [ expression1 ]

then

  statement1

  statement2

  .

  .

elif [ expression2 ]

then

  statement3

  statement4

  .

  .

else

  statement5

fi
```

The case...esac Statement

You 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.

There is only one form of **case...esac** statement which has been described in detail here −

- case...esac statement

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++**

Syntax :

```
case in

Pattern 1) Statement 1;;

Pattern n) Statement n;;

esac
```

while statement

Here command is evaluated and based on the result loop will executed, if command raise to false then loop will be terminated.

Syntax

```
while command
do
   Statement to be executed
done
```

for statement

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

Here var 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 var is set to the next word in the list of words, word1 to wordN.

Syntax

```
for var in word1 word2 ...wordn
do
   Statement to be executed
done
```

<u>**functions**</u>

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**. This is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax −

```
function_name () {
   list of commands
}
```

The name of your function is **function_name**, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example

Following example shows the use of function −

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World"
}

# Invoke your function
Hello
```

Upon execution, you will receive the following output −

```
$./test.sh
Hello World
```