Writing Shell Scripts — The Beginner's Guide

By Muhammad Junaid

Shell scripts are just set of commands that you write in a file and run them together. For anyone who has worked with **DOS's *bat* files**, it's almost the same concept. You just put a series of commands into a text file and run them together. The difference comes from the fact that bash scripts can do a lot more than batch files.

> "A shell script is a computer program designed to be run by the Unix shell, a command-line

> interpreter. The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text."

Unix has more than one possible shell, and scripting any of them is a topic that can easily pack a complete book. In this post, I am going to cover the basic elements of a ***bash script***.

Should I learn?

Agreed that anything you can do with a shell script, you can do that using some programming language such as Ruby, Python or Go but mostly for the small tasks, you will find yourself using Shell Scripts in one way or another.

Shell scripts are used to automate administrative tasks, encapsulate complex configuration details and get at the full power of the operating system. The ability to combine commands allows you to create new commands, thereby adding value to your operating system. Furthermore, combining a shell with graphical desktop environment allows you to get the best of both worlds

- Automate your daily tasks
- Create your own commands with optionally accepting input from the user
- Portability, executing the same script in your mac and your Linux based systems.

Writing Shell Scripts

Let's start by a Hello World example. Open your favorite editor and write a shell script file named as ***my_script.sh*** containing following lines

```
#!/bin/bash
echo "hello world" //print to screen
```

The first line called a **hashbang** or **shebang.** It tells Unix that this script should be run through the **/bin/bash** shell. Second line is just the ***echo*** statement, which prints the words after it to the terminal.

After saving the above file, we need to give it execute permission to make it runnable. You can set the execute permission as follows

```
chmod +x  my_script.sh //add execute permission
```

Execute script as anyone of the following commands

```
$ bash my_script.sh
$ ./my_script.sh
```

Sample Output

```
Hello world
```

Now we are done with the very basic shell script that prints `*Hello world*` to the screen.

Before we go any deeper into few language constructs of shell scripting, you should have some basic knowledge of **Linux commands.** You can find several articles on the internet for that. Here is a sample article showing some of the commonly used ones.

Going Deep

Now that we have seen how to write a basic Hello World example, let's look at some of the language constructs that you will find yourself using most of the time when writing shell scripts.

Variables

To process data, data must be kept in the computer's memory. Memory is divided into small locations, and each location had a unique number called memory address, which is used to hold data.

Programmers can give a unique name to this memory address called ***variables.*** ***Variables*** are a named storage location that may take different values, but only one at a time.

In Linux Shell Scripting, there are two types of variable:

- **System variables —** Created and maintained by Linux itself. This type of variable defined in **CAPITAL LETTERS**.
- **User-defined variables —** Created and maintained by the user. This type of variable defined in ***lower letters.***

System variables can be used in the script to show any information these variables are holding. Like few important **System variables** are:

- **BASH —** *Holds o*ur shell name
- **BASH_VERSION —** Holds our shell version name
- **HOME —** Holds home directory path
- **OSTYPE —** Holds OS type
- **USERNAME –** Holds username who is currently logged in to the machine

NOTE — Some of the above system variables may have a different value in a different environment.

User-defined variables are as simple as we have in any other programming language but variables can store any type of data, as in the following example:

```
# Syntax to define a variable
name=abc
```

To access user-defined variables use the following syntax:

```
#Syntax to access a variable
$name
```

Print to screen:

```
#Syntax to print a variable
echo $name

#output
abc
```

Use the above variable in a string:

```
#Syntax to print a variable
echo "My name is $name"

#output
My name is abc
```

Quotes

Following are the three types of quotes available in Shell scripting.

Double Quotes (") : Anything inside double quotes will be string except \ and **$**. See example

```
#Syntax to define string variable double quotes
str="Shell scripting article"
echo $str

#output
Shell scripting article

#Using \ to escape characters
str="Shell scripting \"article\""
echo $str

#output
Shell scripting "article"

#using variable in a string
user="ABC"
str="Shell scripting \"article\" by $user"
echo $str

#output
Shell scripting "article" by ABC
```

Single quotes (') : Anything inside single quotes will be a string. See example:

```
#Syntax to define string variable using single quotes
str='Shell scripting article'
echo $str

#output
Shell scripting article

#Trying \ to escape characters in single quotes
str='Shell scripting \"article\"'
echo $str

#output
Shell scripting \"article\"
```

Left Quotes (`): Anything enclosed in left quotes will be treated as an executable command. See examples

```
#Syntax to define a string variable
str="Current date is `date`"
echo $str

#output
Current date is Wed Apr 4 10:57:12 +04 2018
```

Conditions [if/else]

Shell scripts use fairly standard syntax for if statements. The conditional statement is executed using either the test command or the [command.

In its most basic form an if statement is:

```
# Syntax of simple if then statement

if [ 35 -gt 0 ]
then
    echo "Greater"
fi

# output
Greater
```

Have you noticed that ***fi*** is just ***if*** spelled backward? See below example that includes an else statement

```
#Syntax of simple if then statement
if [ 35 -gt 45 ]
then
 echo "Greater"
else
 echo "Lesser"
fi

#Output
Lesser
```

Adding an else-if statement structure is used with the ***elif*** command.

```
# Syntax of simple if then else-if statement
if [ 35 -gt 55 ]
then
 echo "Greater"
elif [ 35 -gt 45 ]
then
 echo "Greater"
else
 echo "Lesser"
fi

# Output
Lesser
```

There are many different ways in which conditional statements can be used in Shell scripting. Following tables elaborates on how to add some important comparison:

String Comparisons

Conditions	Description
Str1 = Str2	True if the strings are equal
Str1 != Str2	True if the strings are not equal
-n Str1	True if the string is not null
-z Str1	True if the string is null

Numeric Comparisons

Conditions	Description

```
+------------------+-----------------------------------------------+
| expr1 -eq expr2  | True if the expressions are equal             |
| expr1 -ne expr2  | True if the expressions are not equal         |
| expr1 -gt expr2  | True if expr1 is greater than expr2           |
| expr1 -ge expr2  | True if expr1 is greater than equal to expr2  |
| expr1 -lt expr2  | True if expr1 is less than expr2              |
| expr1 -le expr2  | True if expr1 is less than or equal to expr2  |
| !expr1           | Negates the result of the expression          |
+------------------+-----------------------------------------------+
```

So this is the basic use of conditions in shell scripting is explained.

Looping

Almost all languages have the concept of loops, If we want to repeat a task ten times, we don't want to have to type in the code ten times, with maybe a slight change each time.
As a result, we have **`for`** and **`while`** loops in the shell scripting. This is somewhat fewer features than other languages.

For Loop:

```
# Syntax for simple For loop
for i in 1 2 3 4 5
do
echo "Hello world $i"
done

#Output
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5
```

The above for loop first creates variable ***i*** and assign a number to it from the list of number from 1 to 5, The shell executes echo statement for each assignment of ***i*** and on every ***iteration***, it will echo the statement as shown in the output. This process will continue until the last item.

While Loop

While loop will execute until the condition is true. See below example:

```
# Syntax for simple While loop
i = 1
while [ $i -le 5 ]
do
 echo "Hello world $i"
 i=`expr $i + 1`
done

# Output
Hello world 1
Hello world 2
Hello world 3
Hello world 4
Hello world 5
```

The above script first creates a variable ***i*** with the value ***1***. And then the loop will iterate until the value of ***i*** is less than equals to ***5***. The statement

i=`expr $i + 1` *is responsible for the increment of value of* ***i***.

If this statement is removed the above said loop will be an ***infinite loop***.

Functions

Function is a type of procedure or routine. **Functions** encapsulate a task (they combine many instructions into a single line of code). Most **programming** languages provide many built-in **functions,** that would otherwise require many steps to accomplish, for example calculating the square of a number.

In shell scripting, we can define functions in **two** manners.

1. Creating a function inside the same script file to use.
2. Create a separate file i.e. ***library.sh*** with all useful functions.

See below example to define and use a function in shell scripting:

```
#Syntax to declare a simple function
print_date()
{
echo "Today is `date +"%A %d %B %Y (%r)"`"
return
}

#Calling the above function
print_date
```

```
#Output
Today is Thursday 05 April 2018 (12:11:23 PM)
```

Exit Status

The **exit** command terminates a script, just as in a **C** program. It can also return a value, which is available to the script's parent process.

Every command returns an *exit status* (sometimes referred to as a *return status* or *exit code*). A successful command returns a ***0***, while an unsuccessful one returns a non-zero value that usually can be interpreted as an *error code*. Well-behaved UNIX commands, programs, and utilities return a 0 exit code upon successful completion, though there are some exceptions.

When a script ends with an **exit** that has no parameter, the exit status of the script is the exit status of the last command executed in the script (previous to the **exit**).

```
#Syntax to exit
LAST_COMMAND

# Will exit with the status of the last command.

exit
```

The equivalent of a **exit** is **exit $?** or even just omitting the **exit**.

```
LAST_COMMAND

# Will exit with the status of the last command.

exit $?
```

$? is a special variable in shell that reads the **exit status** of the last command executed. After a function returns, **$?** gives the exit status of the last command executed in the function.