Shell Scripting

A shell script is a file containing a series of commands. The shell reads

this file and carries out the commands as though they have been entered directly on the command

line.

Normally the shell scripts has the file extension .sh

Your default shell is set in the /etc/passwd file, like this line for user Test.

Test:x:510:512:Test ID:/home/Test:/bin/bash

**Comparison between Shells**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Feature | Bourne | Bash | Korn | C | TC |
| Aliases | no | yes | yes | yes | yes |
| Advanced Pattern Matching | no | yes | yes | no | no |
| Command Line Editing | no | yes | yes | no | yes |
| Directory Stacks(pushd,popd) | no | yes | no | yes | yes |
| Filename Completion | no | yes | yes | yes | yes |
| Functions | yes | yes | yes | no | no |
| History | no | yes | yes | yes | yes |
| JobControl | no | yes | yes | yes | yes |
| Key Binding | no | yes | no | no | yes |
| Prompt Formatting | no | yes | no | no | yes |
| Spelling Correction | no | yes | no | no | yes |

**4 Ways of Executing a Shell Script**

* ./scriptfile (or) /path/to/scriptfile
* sh scriptfile
* bash scriptfile
* . ./scriptfile

In order to execute the shell script make sure that you have execute permission. Give execute

permissions as follows.

#chmod +x hello.sh (OR) chmod 111 hello.sh

usually shell script start with **#!/bin/bash**

The first line is called a shebang or a "bang" line. It is nothing but the absolute path to the Bash interpreter. It consists of a number sign and an exclamation point character (#!), followed by the full path to the interpreter such as /bin/bash.

If you do not specify an interpreter line, the default is usually the /bin/sh

**File Naming conventions:**

1) A file name can be a maximum of 255 characters

2) The name may contain alphabets, digits, dots and underscores

3) System commands or Linux reserve words can not be used for file names

4) File system is case sensitive

**comments**

Comments help you keep track of what your script is doing. They are useful because they allow you to leave internal notes throughout your script to remind you of things like what the next command will do or its expected output. You’re essentially annotating your scripts.

There are two types of comments:

* Single-line comment
* Multi-line comment

**single line comment**

A single-line comment starts with a ‘#’ (hash-tag). A line starting with ‘#’ will not get executed.

#Single line comment

echo "hello world"

**Multi-line comment**

There are two ways of writing multi-line comments.

* Using << ‘some\_word’ … ‘some\_word’

eg: << comment

This is line 1.

This is line 2.

This is line 3.

comment

* Using **: ‘…’**

eg: **: ‘**

This is line 1.

This is line 2.

This is line 3.

**’**

**Variables**

Variables are the containers which store data or a useful piece of information as the value inside them. Below is the syntax for a variable: Variable\_name =value

A Variable is a combined form of two words, i.e., vary + able, which means its value can be changed, and it can be used for multiple times.

Variable is known as the temporary storage for any kind of data like integer, float, char, etc. A variable name can include alphabets, digits, and underscore, and its name can be started with alphabets and underscore only.

**Types of Bash Variables**

There are two types of variables in a shell or any UNIX system.

* System-Defined Variables
* User-Defined Variables
* System defined variables

These are predefined variables or the variables which are created and maintained by Linux bash shell. These variables are loaded when you open a new bash session. These variables are used to define system properties. They are defined in capital letters. Let’s now see some of the system-defined variables:

* BASH\_VERSION
* BASH
* PWD
* OSTYPE
* HOME
* LANG
* HOSTNAME
* PATH
* COLUMNS
* USER
* User-defined variables

These are the variables that you define and assign a value.

eg: variable\_name=value

myvar=gfg

**Command Line arguments**

We can pass parameters just after the name of the script while running the bash interpreter command. You can pass parameters or arguments to the file. Just the command for running the script normally by adding the value of the parameters directly to the script. Every parameter is a space-separated value to pass to the shell script.

eg: bash scriptname.sh parameter1 parameter2 parameter3 nth-parameter

|  |  |
| --- | --- |
| **Parameters** | **Function** |
| $1-$9 | Represent positional parameters for arguments one to nine |
| ${10}-${n} | Represent positional parameters for arguments after nine |
| $0 | Represent name of the script |
| $∗ | Represent all the arguments as a single string |
| $@ | Same as $∗, but differ when enclosed in (") |
| $# | Represent total number of arguments |
| $$ | PID of the script |
| $? | Previous cmd execution status (0 – successful, 127 – cmd not found) |

eg: #!/bin/sh

echo "Script Name: $0"

echo "First Parameter of the script is $1"

echo "The second Parameter is $2"

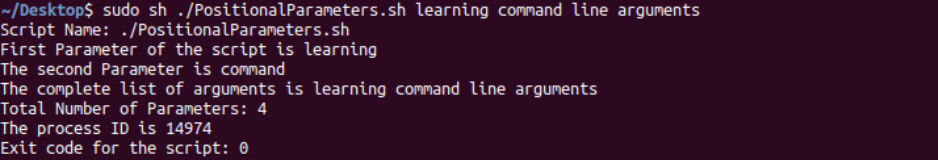
echo "The complete list of arguments is $@"

echo "Total Number of Parameters: $#"

echo "The process ID is $$"

echo "Exit code for the script: $?"

./PositionalParameters.sh learning command line arguments



**String**

String is a group of char enclosed in ‘ ‘ or “ “.

In bash shell, when you use a dollar sign followed by a variable name, shell expands the variable with its value. This feature of shell is called parameter expansion.

But parameter expansion has numerous other forms which allow you to expand a parameter and modify the value or substitute other values in the expansion process. let us review how to use the parameter expansion concept for string manipulation operations.

single='Single quoted'

double="Double quoted"

echo $single

echo $double

---------------------------------------------

string\_var="Shell can be accessed by users using a command line interface"

echo $string\_var

echo ${string\_var}

echo ${#string\_var}

echo ${string\_var:39}

echo ${string\_var:13:8}

echo ${string\_var: -9}

output

Shell can be accessed by users using a command line interface

Shell can be accessed by users using a command line interface

61

command line interface

accessed

interface

**Arithmetic operations**

We use the keyword " expr " to perform arithmetic operations.

a=10

b=3

# there must be spaces before/after the operator

sum=`expr $a + $b`

echo $sum

sub=`expr $a - $b`

echo $sub

mul=`expr $a \\* $b`

echo $mul

div=`expr $a / $b`

echo $div

**User Interaction using read command**

In some cases the script needs to interact with the user and accept inputs.

In shell scripts we use the read statement to take input from the user.

read : read command is used to get the input from the user (Making scripts interactive).