Chapter 9

# Shell Scripts

## What is in This Chapter ?

This chapter discusses shell scripting. **Shell scripts** are small interpreted programs that allow you to examine, manipulate and calculate things at the Linux command level within a Linux shell. There are many basic examples here that make use of the **sh shell scripting language**.



|  |
| --- |
| * 1. **Scripting** |

What is a scripting language? It is a very high-level programming language. Some are general-purpose, some are domain-specific. The languages are limited in that most do not have data types or functions. Most are interpreted, not compiled … which means that they run slower. There are a few different scripting languages out there:

* shell scripting (bash, csh, sh)
* PHP, Perl
* Javascript
* Python
* Ruby
* Lua

Scripting languages are used to do different things, but mainly they are used for:

* performing some kind of one-time task
* automating the execution of other more complex programs
* rapid prototyping

We can use scripting languages to write code that runs in a Linux shell. These are called ***shell scripts***. Typical operations performed by **shell scripts** include:



* system administration tasks
* file manipulation and management
* application configuration and setup
* program execution
* printing text
* testing

A script which sets up the environment, runs the program, and does any necessary cleanup, logging, etc. is called a ***wrapper***.

A shell script is really just a set of commands saved in a file as a program. The commands will depend on the type of shell used, although they are all similar to one another. Here are some shell types:

* Bourne shell (**sh**)
* Bourne-again shell (**bash**)
* C shell (**csh**)
* Korn shell (**ksh**)

We will discuss the Bourne shell (**sh**), which is a subset of **bash**. So, all **sh** commands are valid in **bash**. When we write our script files, they will have a **.sh** file extension.

Here is the Hello World of shell scripting. You can use a text editor to write this script and save it into a file called **helloWorld.sh**:

Indicates that we should always run this script using **sh** rather than **bash** or some other shell.

**#!/bin/sh**

**#This is a comment**

**echo Hello World**

The **#** symbol is used to indicate that the text on that line is a comment (single line only). The **echo** command is used to tell the interpreter to print out the text that follows to the terminal. You run the script by using the **sh** command, followed by the file name:

|  |
| --- |
| **student@COMPBase**:**~**$ sh helloWorld.sh  Hello World  **student@COMPBase**:**~**$ |

You can get user input (in the form of a string) by using the **read** command, followed by a name for the variable that you’d like to store the string in. We can then refer to the value of that variable anywhere in the script by using the variable name with a **$** character in front. Here is a script that gets the user’s name and then displays it:

**#!/bin/sh**

**#This script asks for the user’s name**

**echo What is your name?**

**read NAME**

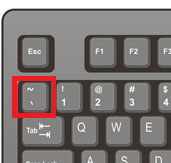
**echo $NAME is a cool name**

|  |
| --- |
| **student@COMPBase**:**~**$ sh getName.sh  What is your name?  **Mark**  Mark is a cool name  **student@COMPBase**:**~**$ |

It is interesting that shell script variables do not need to be declared. Space is allocated the first time the shell sees a new variable. Notice as well that no data types are specified since all variables are stored as strings. The variable should be in uppercase letters with underscores when needed.

We can even hardcode constants, which are treated as variables. However, we need to make sure that there is no space before the equal sign. Also, the value can be anything, but if we want to use the values in a numerical expression, they can only be integers (i.e., no floats).

**COUNT=5 #no spaces before or after =**

**COUNT = 5 #this won’t work!!**

To calculate a math expression, we need to use the **expr** command, which is a little cumbersome to work with. It is actually an external program that we will run. First of all, the math expression must be encapsulated with a single backquote character **`** … which is not the usual single straight quote character **'**. The backquote character may be hard to find. It is under the ESC key on my keyboard.

The **expr** command can take in some variables (use the **$** in front of the name) as well as some math operators. You should precede the math operators with a **\** character. Here is an example of how to use it. This script asks for the number of bags of milk and cartons of eggs to buy and then calculates the integer-based price:

**#!/bin/sh**

**#Calculate price for buying X bags of milk and Y cartons of eggs**

**MILK=4 #Notice no spaces before or after the = sign**

**EGGS=2**

**echo How many bags of milk do you want?**

**read NUM\_MILK**

**echo How many cartons of eggs do you want?**

**read NUM\_EGGS**

**PRICE=`expr $NUM\_MILK \\* $MILK \+ $NUM\_EGGS \\* $EGGS`**

**echo The total price for $NUM\_MILK bag\(s\) of milk and $NUM\_EGGS carton\(s\) of eggs is \$$PRICE**

|  |
| --- |
| **student@COMPBase**:**~**$ sh prices.sh  How many bags of milk do you want?  **1**  How many cartons of eggs do you want?  **1**  The total price for 1 bag(s) of milk and 1 carton(s) of eggs is $6  **student@COMPBase**:**~**$ sh prices.sh  How many bags of milk do you want?  **3**  How many cartons of eggs do you want?  **5**  The total price for 3 bag(s) of milk and 5 carton(s) of eggs is $22  **student@COMPBase**:**~**$ |

Notice the use the backslash character in the last echo command line. It is used to display a special character, which is the parenthesis or dollar sign in this case. We can even supply values to the script from the command line. We access the command line arguments by using **$1**, **$2**, **$3**, etc. You can use **$0** to get the command itself (i.e., the name of the script file), **$#** to get the number of arguments and **$$** to get the process ID. Here is a modified program that uses command line arguments instead of asking the user for the numbers:

**#!/bin/sh**

**#Calculate price for buying X bags of milk and Y cartons of eggs**

**# X and Y are supplied as command line arguments**

**echo The command is $0**

**echo There are $# command line arguments**

**echo The process ID is $$**

**MILK=4**

**EGGS=2**

**PRICE=`expr $1 \\* $MILK \+ $2 \\* $EGGS`**

**echo The total price for $1 bag\(s\) of milk and $2 carton\(s\) of eggs is \$$PRICE**

|  |
| --- |
| **student@COMPBase**:**~**$ sh cmdLine.sh **1** **1**  The command is cmdLine.sh  There are 2 command line arguments  The process ID is 3467  The total price for 1 bag(s) of milk and 1 carton(s) of eggs is $6  **student@COMPBase**:**~**$ sh cmdLine.sh **3** **5**  The command is cmdLine.sh  There are 2 command line arguments  The process ID is 3469  The total price for 1 bag(s) of milk and 1 carton(s) of eggs is $22  **student@COMPBase**:**~**$ |

We can also do a **FOR** loop in which we need to specify a set of strings to loop through:

**#!/bin/sh**

**#This example just shows that we can loop through strings**

**for i in 1 2 5 A Z temp output**

Notice how we just list all strings here.

**do**

**echo file$i.txt #or can use file${i}.txt**

**done**

|  |
| --- |
| **student@COMPBase**:**~**$ sh forLoop.sh  file1.txt  file2.txt  file5.txt  fileA.txt  fileZ.txt  filetemp.txt  fileoutput.txt  **student@COMPBase**:**~**$ |

Here is an example of a **WHILE** loop that repeats until the user enters a string other than **yes**:

**#!/bin/sh**

**#Example showing how to use a while loop**

**RESPONSE=yes**

The spacing is required here!!

**while [ $RESPONSE = yes ]**

**do**

**echo Do you want to loop again?**

**read RESPONSE**

**done**

**echo Goodbye!**

|  |
| --- |
| **student@COMPBase**:**~**$ sh whileLoop.sh  Do you want to loop again?  yes  Do you want to loop again?  yes  Do you want to loop again?  no  Goodbye!  **student@COMPBase**:**~**$ |

Of course, sooner or later we will need the ability to make decisions. There are **if/then/fi** and **if/then/else/fi** control structures for doing this.

**#!/bin/sh**

**#This script tests out the IF statement based on a command line name**

**if [ $1 = Mark ]; then**

The spacing is very important here!!

**echo Hello Mark**

**else if [ $1 = Christie ]; then**

**echo Hello Christie**

Semicolon required!!

**else**

**echo I do not know you**

**fi**

Backwards “if” to end it all

**fi**

|  |
| --- |
| **student@COMPBase**:**~**$ sh if.sh Mark  Hello Mark  **student@COMPBase**:**~**$ sh if.sh Christie  Hello Christie  **student@COMPBase**:**~**$ sh if.sh Bob  I do not know you  **student@COMPBase**:**~**$ |

Within the square brackets, we can do various types of testing of conditionals:

* **-z** tests if string is empty
* **-n** tests if string is not empty
* **-lt**, **-le** tests whether LHS operand is < or <= RHS operand
* **-gt**, **-ge** tests whether LHS operand is > or >= RHS operand

Here is an example showing some nested **IF** statements that make use of the **-n** and **-lt** conditions by taking in three command-line integers and displaying the smallest one:

**#!/bin/sh**

**#This script tests out conditionals based on 3 command line integers**

**if [ -z $3 ]; then**

**echo ERROR I need three numbers**

**else**

** if [ $1 -lt $2 ]; then**

**if [ $1 -lt $3 ]; then**

**echo $1 is the smallest number**

**else**

**echo $3 is the smallest number**

**fi**

**else**

**if [ $2 -lt $3 ]; then**

**echo $2 is the smallest number**

**else**

**echo $3 is the smallest number**

**fi**

**fi**

**fi**

|  |
| --- |
| **student@COMPBase**:**~**$ sh tests.sh **2 4 6**  2 is the smallest number  **student@COMPBase**:**~**$ sh tests.sh **7 3 9**  3 is the smallest number  **student@COMPBase**:**~**$ sh tests.sh **8 5 1**  1 is the smallest number  **student@COMPBase**:**~**$ sh tests.sh **34 64**  ERROR I need three numbers  **student@COMPBase**:**~**$ sh tests.sh **34**  ERROR I need three numbers  **student@COMPBase**:**~**$ |

Here are some more options for the conditional statements, which are file-related. These are very handy as many shell scripts are written to manipulate and examine files:

* **-f** tests whether a file with the given name exists
* **-d** tests whether a given operand is a directory
* **-r**, **-w**, **-x** tests whether the given file has read/write/execute permissions

Here is a script that looks for a particular file or directory (specified in the command line) and then indicates whether the file was found and displays its read/write/execute permissions.

**#!/bin/sh**

**#This script checks if a file or directory exists (specified as command**

**#line argument). It also checks the permissions**

**if [ $# = 0 ]; then**

**echo Error\: Missing Filename**

** echo USAGE\: sh fileCheck.sh \<fileName\>**

**exit**

Quit the script

**fi**

**if [ -f $1 ]; then**

**echo FILE \"$1\" exists**

**if [ -r $1 ]; then echo FILE is readable**

**fi**

**if [ -w $1 ]; then echo FILE is writeable**

**fi**

**if [ -x $1 ]; then echo FILE is executable**

**fi**

**else**

**if [ -d $1 ]; then**

**echo DIRECTORY \"$1\" exists**

**if [ -r $1 ]; then echo DIRECTORY is readable**

**fi**

**if [ -w $1 ]; then echo DIRECTORY is writeable**

**fi**

**if [ -x $1 ]; then echo DIRECTORY is executable**

**fi**

**else**

**echo File/Directory \"$1\" not found**

**fi**

**fi**

|  |
| --- |
| **student@COMPBase**:**~**$ ls  **aDirectory** fileCheck.sh getName.sh helloWorld.sh prices.sh whileLoop.sh  cmdLine.sh forLoop.sh **hello** if.sh tests.sh  **student@COMPBase**:**~**$ sh fileCheck.sh **getName.sh**  FILE "getName.sh" exists  FILE is readable  FILE is writeable  **student@COMPBase**:**~**$ sh fileCheck.sh **hello**  FILE "hello" exists  FILE is readable  FILE is writeable  FILE is executable  **student@COMPBase**:**~**$ sh fileCheck.sh **aDirectory**  DIRECTORY "aDirectory" exists  DIRECTORY is readable  DIRECTORY is writeable  DIRECTORY is executable  **student@COMPBase**:**~**$ sh fileCheck.sh **missing**  File/Directory "missing" not found  **student@COMPBase**:**~**$ sh fileCheck.sh  Error: Missing Filename  USAGE: sh fileCheck.sh <fileName>  **student@COMPBase**:**~**$ |

We have covered the basics of the SH scripting language but there are external shell programs that we can also invoke. That is, we can use any Unix commands or executable user programs from within our script. This is where the power really lies in scripting. We have seen one use of this already with the **expr** command, which is actually an external program being run. As with the **expr** command, all external program calls must be encapsulated with a single backquote character **`**.

Here is an example of a script that uses the **ls** Unix command to get the files in the current directory and then shows them in two ways:

**#!/bin/sh**

**#This script lists all files in two ways**

**echo Here is the result of executing the \"ls\" command\:**

**echo**

Stores result of “ls” command in string called FILES.

**FILES=`ls`**

**echo $FILES**

**echo**

**echo Here are the files one at a time\:**

**echo**

**for i in $FILES**

**do**

**echo Filename $i**

**done**

|  |
| --- |
| **student@COMPBase**:**~**$ sh listFiles.sh  Here is the result of executing the "ls" command:  aDirectory cmdLine.sh fileCheck.sh forLoop.sh getName.sh hello helloWorld.sh if.sh listFiles.sh listFiles.sh~ prices.sh tests.sh whileLoop.sh  Here are the files one at a time:  Filename aDirectory  Filename cmdLine.sh  Filename fileCheck.sh  Filename forLoop.sh  Filename getName.sh  Filename hello  Filename helloWorld.sh  Filename if.sh  Filename listFiles.sh  Filename listFiles.sh~  Filename prices.sh  Filename tests.sh  Filename whileLoop.sh  **student@COMPBase**:**~**$ |

This script counts files and directories:

**#!/bin/sh**

**#This script counts the files and directories**

**FILES=`ls`**

**FILE\_COUNT=0**

**DIR\_COUNT=0**

**for file in $FILES**

**do**

**if [ -f $file ]; then**

**FILE\_COUNT=`expr $FILE\_COUNT + 1`**

**fi**

**if [ -d $file ]; then**

**DIR\_COUNT=`expr $DIR\_COUNT + 1`**

**fi**

**done**

**echo There are $FILE\_COUNT files and $DIR\_COUNT directories**

|  |
| --- |
| **student@COMPBase**:**~**$ ls  **aDirectory** fileCheck.sh **hello** listFiles.sh whileLoop.sh  cmdLine.sh forLoop.sh helloWorld.sh prices.sh  countFiles.sh getName.sh if.sh tests.sh  **student@COMPBase**:**~**$ sh countFiles.sh  There are 12 files and 1 directories  **student@COMPBase**:**~**$ |

The **tr** program allows us to replace characters in a string with other characters. We supply two parameters … the string to look for and the string to replace it with. It takes data from **stdin** and outputs to **stdout**. We can use a pipeline to do the conversion. Here is a simple example:

**#!/bin/sh**

**#This script does a search and replace for a string**

**STR="This sentence is about to be modified."**

**OUT=`echo $STR | tr e o`**

Pipe the string as input to the **tr** program

**echo $OUT**

|  |
| --- |
| **student@COMPBase**:**~**$ sh replace.sh  This sontonco is about to bo modifiod.  **student@COMPBase**:**~**$ |

This can be powerful when we combine it with file searches to rename a bunch of files. Here is an example of a program that renames all the files in the current directory to uppercase:

**#!/bin/sh**

**#This script renames all files to uppercase**

Can use [:upper:] or [:lower:] to indicate an uppercase or lowercase character

**FILES=`ls`**

**for NAME in $FILES**

**do**

** NEW\_NAME=`echo $NAME | tr [:lower:] [:upper:]`**

**if [ $NEW\_NAME != $NAME ]; then**

**mv $NAME $NEW\_NAME**

**fi**

**done**

|  |
| --- |
| **student@COMPBase**:**~**$ ls  rename.sh test01.dat test03.dat test05.dat test07.dat  rename.sh~ test02.dat test04.dat test06.dat test08.dat  **student@COMPBase**:**~**$ sh rename.sh  **student@COMPBase**:**~**$ ls  RENAME.SH TEST01.DAT TEST03.DAT TEST05.DAT TEST07.DAT  RENAME.SH~ TEST02.DAT TEST04.DAT TEST06.DAT TEST08.DAT  **student@COMPBase**:**~**$ |

The **cut** program allows us to remove portions of a line of input. We can use the **-d** option to indicate a delimiter in the string. This indicates the character (or string) that separates the tokens (e.g., words) of the string. We may often use " " as the delimiter string.

We can use the **-f n** option (where **n** is a number starting at 1) to indicate that the **n**th token (i.e., word) of each line should be retained. We can list some tokens by using a comma between the token numbers that we want like this **-f n1,n2,n3**.

Here is an example that keeps the 2nd and 5th word from each sentence:

**#!/bin/sh**

**#This script maintains the 2nd and 5th word from each sentence**

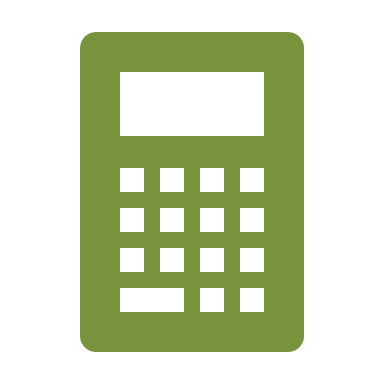
**STR="The cat climbed the tree\nThe dog chased the stick\nThe turtle took a nap"**

**OUT=`echo $STR | cut -d " " -f 2,5`**

**echo $OUT**

|  |
| --- |
| **student@COMPBase**:**~**$ sh cut.sh  cat tree dog stick turtle nap  **student@COMPBase**:**~**$ |

Another useful option is the **-b** option. It allows you to extract a range of characters from a string based on the number of the character in the string. So, we could use this, for example, along with the **ls -l** Unix command to get the total of all file sizes in a directory:

****

**#!/bin/sh**

**#This script uses ls -l and the examines the**

**#output to calculate the total files sizes**

**SIZES=`ls -l | cut -b 30-34`**

**TOTAL=0**

**for SIZE in $SIZES**

**do**

**TOTAL=`expr $TOTAL + $SIZE`**

**done**

**echo Total of all file sizes = $TOTAL bytes**

|  |
| --- |
| **student@COMPBase**:**~**$ ls -l  total 72  drwxrwxr-x 2 student student 4096 Jul 26 12:50 **aDirectory**  -rw-rw-r-- 1 student student 354 Jul 24 15:18 cmdLine.sh  -rw-rw-r-- 1 student student 318 Jul 26 11:50 countFiles.sh  -rw-rw-r-- 1 student student 198 Jul 26 13:01 cut.sh  -rw-rw-r-- 1 student student 234 Jul 26 13:08 cutSizes.sh  -rw-rw-r-- 1 student student 740 Jul 26 11:13 fileCheck.sh  -rw-rw-r-- 1 student student 163 Jul 24 15:36 forLoop.sh  -rw-rw-r-- 1 student student 110 Jul 24 15:18 getName.sh  -rwxrwxr-x 1 student student 7348 Jul 26 11:18 **hello**  -rw-rw-r-- 1 student student 52 Jul 24 15:19 helloWorld.sh  -rw-rw-r-- 1 student student 223 Jul 24 16:27 if.sh  -rw-rw-r-- 1 student student 232 Jul 26 11:42 listFiles.sh  -rw-rw-r-- 1 student student 404 Jul 24 15:19 prices.sh  -rw-rw-r-- 1 student student 213 Jul 26 12:52 rename.sh  -rw-rw-r-- 1 student student 147 Jul 26 12:00 replace.sh  -rw-rw-r-- 1 student student 413 Jul 24 16:57 tests.sh  -rw-rw-r-- 1 student student 164 Jul 24 15:39 whileLoop.sh  **student@COMPBase**:**~**$ sh cutSizes.sh  Total of all file sizes = **15409** bytes  **student@COMPBase**:**~**$ |

You can use **cut –-help** in the shell window to see all the options.

There are even more sophisticated string processing programs and Linux utilities:

* **awk** – provides powerful formatting capabilities similar to **printf()**.
* **basename** – strips the directory and (optionally) suffix from a filename.
* **find** – finds files in and below directories according to their name, age, size, etc..
* **grep** – searches text for matching lines based on regular expressions.
* **sed** – an advanced version of **tr** which can be used for search and replace.
* **sort** – can be used to sort strings in various ways.
* **test** – check existence of a file and its permissions; or compare numbers and strings.
* **wc** – counts words, lines and characters.

Check out the **man** pages on these. You can do some really amazing things. Remember … through pipelining, you can join/merge many programs together to produce a very sophisticated shell script.

