Operating Systems Lab FALL 2024

**Lab Manual 02**

**Shell Scripting**

**SHELL SCRIPTING**

If the shell offers the facility to read its commands from a text file ,then its syntax and features may be thought of as a programming language, and such files may be thought of as scripts. So a shell is actually two things:

1. An interface between user and OS.
2. A programming language.

Shell Script is series of commands written in plain text file the shell reads the commands from the file just as it would have typed them into a terminal. The basic advantage of shell scripting includes:

1. Shell script can take input from user, file and output them on screen.

1. Useful to create our own commands.

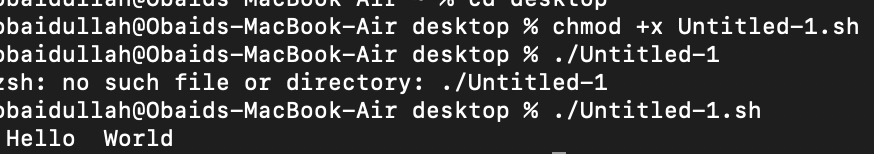
1. Save lots of time.

1. To automate some task of day today life.

1. System Administration part can be also automated.

## EXAMPLES

|  |  |
| --- | --- |
| *# ! / bin / bash*  *# F i r s t s h e l l s c r i p t* clear  **echo** " Hello World" |  |



Use any editor (gedit etc) to write shell script. Then save the shell script to a directory and name it *intro*. Shell scripts don’t need a special file extension, so leave the extension blank (or you can add the extension *.sh*).

## FILE MODES AND PERMISSIONS

Every Linux file has a set of permissions that determine whether you can read, write, or run the file. Running ls -l displays the permissions. The file’s mode represents the file’s permissions and some extra information. There are four parts to the mode, as illustrated in Figure1 .

A diagram of a user

Description automatically generated

Figure 1: File Mode

The first character of the mode is the file type. A dash (-) in this position, as in the example, denotes a regular file, meaning that there’s nothing special about the file. This is by far the most common kind of file. Directories are also common and are indicated by a d in the file type slot. The rest of a file’s mode contains the permissions, which break down into three sets: user, group, and other, in that order. For example, the rwcharacters in the example are the user permissions, the r– characters that follow are the group permissions, and the final r– characters are the other permissions.

|  |  |
| --- | --- |
| File Mode | Explanation |
| r | Means that the file is readable. |
| w | Means that the file is writable. |
| x | Means that the file is executable (you can run it as a program). |
| - | Means nothing |

The user permissions (the first set) pertain to the user who owns the file. The second set, group permissions, are for the file’s group (somegroup in the example). Any user in that group can take advantage of these permissions. Everyone else on the system has access according to the third set, the other permissions.

## MODIFYING PERMISSIONS

To execute a script first we will make it executable. To change permissions, use the chmod command. Only the owner of a file can change the permissions. There are two ways to write the chmod command:

1. *chmod {a,u,g,o}{+,-}{r,w,x}*

{all, user, group, or other}, {read, write, and execute} ’+’ means add permission and ’-’ means remove permission.

For example, to add group (g) and others (o) read (r) permissions to file, you could run these two commands:

$ chmod g+r file

$ chmod o+r file

Or you could do it all in one shot:

$ chmod go+r file

To remove these permissions, use *go-r* instead of *go+r*.

1. *chmod* h3*DigitNumber*i

Every digit sets permission for the owner(user), group and others as shown in Table 2.

To set the permission decipher the number first and then execute the command as

$ chmod 700 file

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| User | |  | Group | |  | Other | |  | 3 Digit No. | Description |
| r | w | x | r | w | x | r | w | x |  |  |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 400 | user: read |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 040 | group: read |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 711 | user: read/write/execute; group, other: execute |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 644 | user: read/write; group, other: read |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 600 | user: read/write; group, other: none |

Table 1: Examples of chmod with 3 Digit numbers

## RUNNING SHELL SCRIPTS

Execute your script as

./script-name

1. **IMPORTANT POINTS**

Following are some important points while writing a script:

* 1. # is used to write comments in your script

* 1. Whenever a semicolon (;) is placed between two commands, shell will treat them as separate commands. So if you want to write all the commands in your shell script in one line place a semicolon in between them.

* 1. A script may start with the line *#!/bin/bash* to tell your interactive shell that the program which follows should be executed by bash.

**VARIABLES IN SHELL**

In Linux shell there are three types of variables:

* 1. User defined or shell variables

* 1. System or Environment variables

* 1. Parametric variables

## USER DEFINED OR SHELL VARIABLES

The shell can store temporary variables, called shell variables, containing the values of text strings. Shell variables are very useful for keeping track of values in scripts. To assign a value to a shell variable, use the equal sign (=). Here’s a simple example:

$ STUFF=blah

The preceding example sets the value of the variable named STUFF to blah. To access this variable, use

*$STUFF* (for example, try running echo *$STUFF*).

Rules for defining variables:

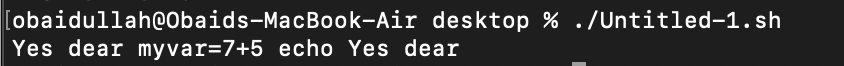
1. Variable name must begin with Alphanumeric character or underscore character, followed by one or more Alphanumeric character.
2. Don’t put spaces on either side of the equal sign when assigning value to variable.

1. Variables are case-sensitive, just like filename in Linux. 4. You can define NULL variable as

var= var=""

5. Do not use ?,\* etc, to name your variable names.

|  |
| --- |
| *# ! / bin / bash s c r i p t*  *#*  *# v a r i a b l e s in s h e l l*  *#* myvar=Hello **echo** $myvar  myvar=" Yes dear "  **echo** $myvar myvar=7+5 **echo** $myvar |



Notice in last assignment, myvar is assigned the string "7+5" and not the result i.e 12.

## SYSTEM OR ENVIRONMENT VARIABLES

An environment variable is like a shell variable, but it’s not specific to the shell. All processes on Unix systems have environment variable storage. The main difference between environment and shell variables is that the operating system passes all of your shell’s environment variables to programs that the shell runs, whereas shell variables cannot be accessed in the commands that you run. Assign an environment variable with the shell’s export command. For example, if you’d like to make the *$STUFF* shell variable into an environment variable, use the following:

$ STUFF=blah

$ export STUFF

Environment variables are useful because many programs read them for configuration and options. Listed below are some common environment variables:

*1. $PATH*

*PATH* is a special environment variable that contains the command path (or path for short). A command path is a list of system directories that the shell searches when trying to locate a command. For example, when you run *ls*, the shell searches the directories listed in *PATH* for the ls program. If programs with the same name appear in several directories in the path, the shell runs the first matching program. If you run

$ echo $PATH

you’ll see that the path components are separated by colons (:). For example:

$ echo $PATH

will output */usr/local/bin:/usr/bin:/bin*

To tell the shell to look in more places for programs, change the PATH environment variable. For example, by using this command, you can add a directory dir to the beginning of the path so that the shell looks in dir before looking in any of the other PATH directories.

$ PATH=dir:$PATH

Or you can append a directory name to the end of the PATH variable, causing the shell to look in dir last:

$ PATH=$PATH:dir

1. *$HOME*

The home directory of the current user.

1. *$IFS*

An input field separator; a list of characters that are used to separate words when the shell is reading input, usually space, tab and newline characters.

## PARAMETRIC VARIABLES

These variables keep the values of the command line arguments passed to the Scripts. Most shell scripts understand command-line parameters and interact with the commands that they run. These parametric variable passed to the script make the code more flexible. These variables are like any other shell variable like Environment and Shell Variables, except that you cannot change the values of certain ones. Following are some parametric variables and a set of environment variables used to handle parametric variables.

**1. Individual Arguments: $1, $2, ...**

$1, $2, and all variables named as positive nonzero integers contain the values of the script parameters, or arguments. For example, say the name of the following script is pshow:

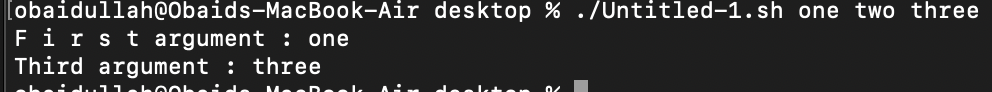
# ! / bin / bash

echo F i r s t argument : $1

echo Third argument : $3

Try running the script as follows to see how it prints the arguments:

$ ./pshow one two three



Output looks like:

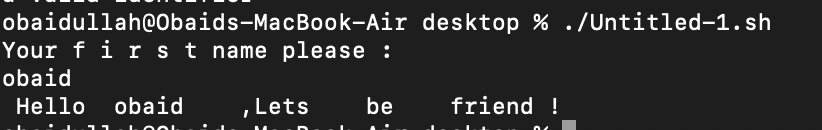
*First argument: one*

*Third argument: three*

**11 INPUT VALUES**

The read command is used to get a line of input into a variable.

|  |
| --- |
| # ! / bin / bash  #  # Script to read your name from key−board  echo "Your f i r s t name please : "  read fname  echo " Hello $fname ,Lets be friend ! " |



Rules to input variables:

1. Each argument must be a variable name without the leading "$".

1. The built in command reads a line of input and separates the line into individual words using the "IFS" inter field separator.
2. Each word in the line is stored in a variable from left to right.

1. The first word is stored in the first variable, the second word to the second variable and so on.
2. If there are fewer variables than words, then all remaining words are then assigned to the last variable.
3. If you have more variables than words defined, then any excess variables are set to null.

|  |  |
| --- | --- |
| # ! / bin / bash  #  # Script to read your name from key−board  read first middle last  echo " Hello $first $middle $last " |  |
| 12 **CONDITIONAL STATEMENTS** | |

The Bourne shell has special constructs for conditionals, such as

1. If statement

1. If-else statement

1. If-elif statement

1. Case Statement

## IF STATEMENT

The basic syntax of if statements is:

|  |  |  |
| --- | --- | --- |
| *# ! / bin / bash* **if** [ conditional  **then** statement1 statement2 **fi** | expression | ] |

## TESTING CONDITION

test command or [ expr ]

are used to see if an expression is true, and if it is true it return zero(0) otherwise returns nonzero for false.

The folowing script determine whether given number is equal to 100 or not.

# ! / bin / bash

#

# Script to see whether argument count=100

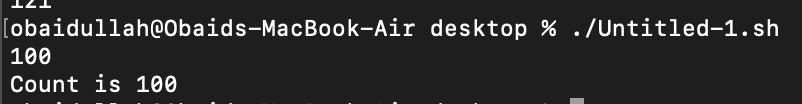
read count

if [ $count -eq 100 ]

then

echo "Count is 100 "

fi



The following script determine whether given argument number is positive using *test*.

|  |
| --- |
| # ! / bin / bash  #  # Script to see whether argument count=100  read count  if test $count -gt 0  then  echo "Count is positive"  fi |

### 1. Mathematical Operators

Figure 2 shows different conditions that we can use to test our expressions. It’s important to recognize that the equal sign (=) looks for string equality, not numeric equality. Therefore, [ 1 = 1 ] returns 0 (true), but [ 01 = 1 ] returns false. When working with numbers, use -eq instead of the equal sign: [ 01 -eq 1 ] returns true.

A table with numbers and symbols

Description automatically generated

Figure 2: Conditions for test and expressions

### 2. String comparisons

Following script checks whether the script’s first argument is "hi" where a string comparison is made.

# ! / bin / bash

if [ $1 = "hi" ]

then

echo "The first argument was hi "

fi

There is a slight problem with the condition in preceding example due to a very common mistake: $1 could be empty, because the user might not enter a parameter. Without a parameter, the test reads *[ = hi ]*, and the command aborts with an error. You can fix this by enclosing the parameter in quotes::

if [ "$1" = hi ]; then

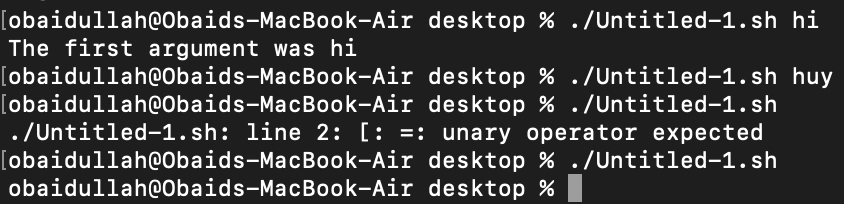
# ! / bin / bash

if [ "$1" = "hi" ]

then

echo "The first argument was hi "

fi



|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| string1 = string2 | string1 is equal to string2 |
| string1 != string2 | string1 is NOT equal to string2 |
| -n string1 | string1 is NOT NULL and does exist |
| -z string1 | string1 is NULL and does exist |

Table 2: String Comparisons

### 3. Test for file and directory

Most file tests that are commonly used are unary operations because they require only one argument: the file to test.

#!/bin/bash

somefile=a.txt

if [ -r $somefile ] ; then

content=$(cat $somefile)

echo "$content"

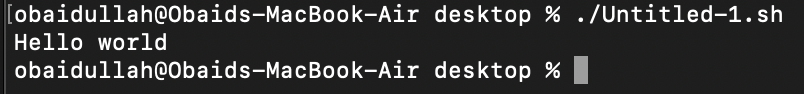
elif [ -f $somefile ] ; then

echo "The file somefile exists but is not readable to the script"

else

echo "The file somefile does not exist"

fi



In the above script first we check if the file somefile is readable. If so, we read it into a variable. If not, we check if it actually exists. If that’s true, we report that it exists but isn’t readable otherwise it moves to the else condition.

|  |  |
| --- | --- |
| **Test** | **Meaning** |
| -s file | Non empty file |
| -f file | Is file exist or normal file and not a directory |
| -d dir | Is directory exist and not a file |
| -w file | Is writable file |
| -r file | Is read-only file |
| -x file | Is executable file |

Table 3: Test for File and Directory

### 4. Logical Operators

You can invert a test by placing the ! operator before the test arguments. For example, *[ ! -f file ]* returns true if file is not a regular file. Furthermore, the -a and -o flags are the logical *and* and *or* operators. For example, *[ -f file1 -a file2 ]*.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| ! expression | Logical NOT |
| expression1 -a expression2 | Logical AND |
| expression1 -o expression2 | Logical OR |

Table 4: Logical Operators

## IF ELSE STATEMENT

The basic syntax of if-else statements is:

|  |  |  |
| --- | --- | --- |
| **if** condition  **then** execute **else**  **else** execute all commands upto fi  **fi** | a l l | commands upto |

Examples using if-else

# ! / bin / bash

echo " Is it morning? Please answer yes or no"

read timeofday

if [ $timeofday = "yes" ] ;

then echo "Good morning"

else echo "good afternoon "

fi

## 

# ! / bin / bash

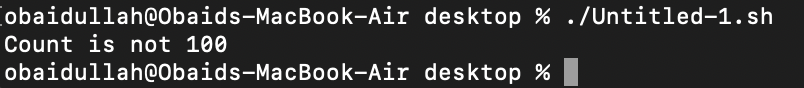
count=99

if [ $count -eq 100 ]

then echo "Count is 100 "

else echo "Count is not 100 "

fi



# !/bin/bash

#script to see whether argument is positive or negative

if [ $# -eq 0 ]

then

echo "$0:You must supply one integer"

exit 1

fi

if test $1 -gt 0

then

echo "$1 is positive number"

else

echo "$1 is negative"

fi

## 

## IF-ELIF STATEMENT

The basic syntax of if-elif statements is:

|  |
| --- |
| **if** condition  **then**  . . . . . . . . . . . . .  **elif** condition  **then**  . . . . . . . . . . . . .  **else**  . . . . . . . . . . . . .  **fi** |

Example:

#!/bin/bash

echo "Is it morning? Please answer yes or no"

read timeofday

if [ "$timeofday" = "yes" ]; then

echo "Good morning"

elif [ "$timeofday" = "no" ]; then

echo "Good afternoon"

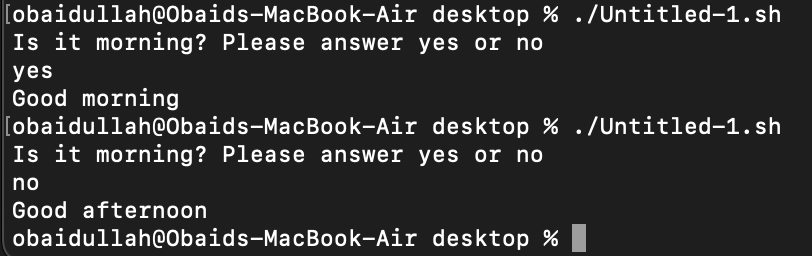
else

echo "Sorry, $timeofday not recognized."

exit 1

fi

exit 0



|  |
| --- |
| 13 **ARITHMETIC OPERATIONS** |

Arithmetic expansion and evaluation is done by placing an integer expression using the following format:

$((expression))

$(( n1+n2 ))

The shell script below includes some arithmetic operations.

#!/bin/bash

clear

echo "hello world"

var1=10

var2=20

var3=$(( var1 + var2 ))

var4=$(( var1 - var2 ))

var5=$(( var1 \* var2 ))

var6=$(( var1 / var2 ))

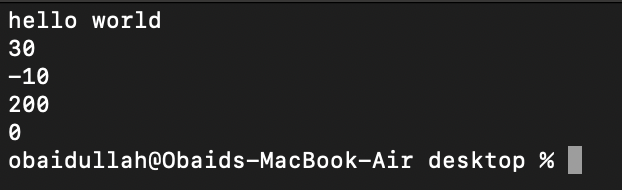
echo $var3

echo $var4

echo $var5

echo $var6

exit 0



There are two kinds of loops in the shell: for and while loops.

### 1. For Loop

The basic syntax of for loop is:

|  |  |  |  |
| --- | --- | --- | --- |
| **for** variable\_name **in** { l i s t of values } **do** execute one **for** each item **in** the l i s t **until** repeat a l l statement between **do** and **done**  **done** l i s t | is | not | finished |

Examples:

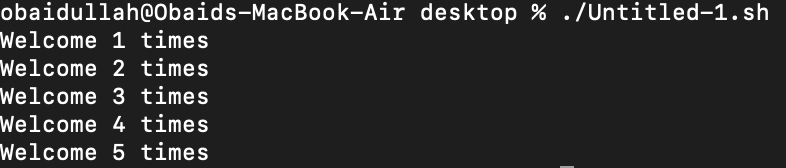
#!/bin/bash

for i in 1 2 3 4 5

do

echo "Welcome $i times"

done

**

#!/bin/bash

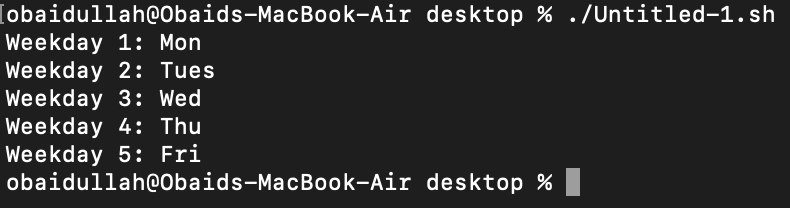
i=1

for day in Mon Tues Wed Thu Fri

do

echo "Weekday $(( i++ )): $day"

done



### 2. While Loop

The basic syntax of while loop is:

|  |  |
| --- | --- |
| *# ! / bin / bash* **while** [ condition  **do** command\_1 command\_2  . . . . . . . . . . . . . .  command\_N **done** | ] |

Examples:

#!/bin/bash

echo "Enter password"

read trythis

while [ $trythis != "secret" ]

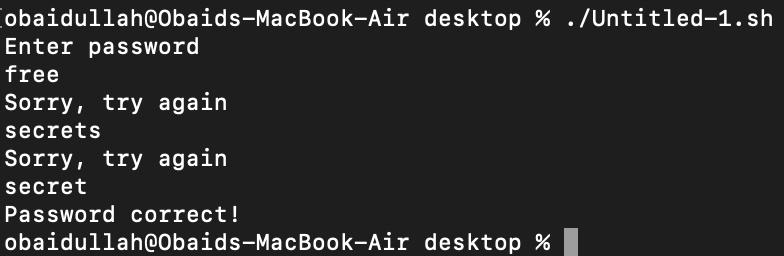
do

echo "Sorry, try again"

read trythis

done

echo "Password correct!"



#!/bin/bash

# Set n to 1

n=1

# Continue until n equals 5

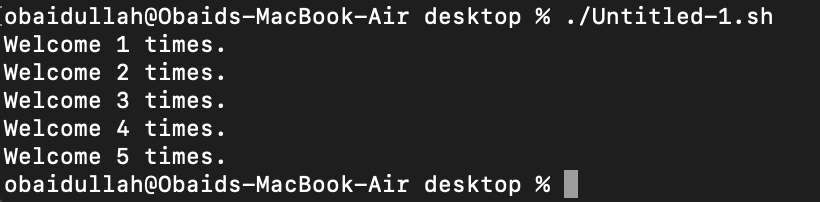
while [ $n -le 5 ]

do

echo "Welcome $n times."

n=$(( n+1 ))

done



#!/bin/bash

counter=$1

factorial=1

while [ $counter -gt 0 ]

do

factorial=$(( factorial \* counter ))

counter=$(( counter - 1 ))

done

echo $factorial

