

# Shell Programming I<sup>1</sup>

*This is a self-paced tutorial sheet.*

## What is “Shell Programming”?

In Linux, it is sometimes necessary to perform a number of complex shell commands in order to achieve an overall goal. In the situation where we have to perform a sequence of complex or repetitive shell commands, this process can be very tedious and time consuming. The interpretative nature of the shell in Linux provides us with an alternative that helps speeds up this process. This facility is known as “Shell Scripting” or “Shell Programming”. A script is a file containing shell commands that can interact with the user, repeat commands and make decisions.

To avail of this flexibility provided by “Shell Programming”, all we need to do is to place the shell commands (which we would normally type at the command prompt) into a file and in turn execute the file.

### Example 1

Suppose we need to find out the number of logged-in users several times throughout the day. The Linux command would be as follows:

`$ who | wc -l` → Note this is letter ‘l’ and not number ‘1’

(N.B. Do not type the \$ - this is just shorthand for the prompt)

Note that this command is actually a combination of two commands, connected by a **pipe**. A pipe is used to connect the *output* of one command to the *input* of another command. The vertical bar symbol (|) is used to specify a pipe. In this example, the output of **who** is provided as input to the **wc -l** command. **who** lists the users currently logged on, one per line. **wc -l** counts the number of lines in its input.

So, to avoid having to re-type this command each time we wish to find out how many users are logged in, we will create a file called “numusers”.

1. We create the file:  
`$ nano numusers`
2. We type in the shell command  
`who | wc -l`
3. save and exit the file  
(*CTRL-O to save and CTRL-X to exit*)  
(i.e. hold down *CTRL* key and press *O*, etc)

**nano** is a command-line text editor.  
**gedit** is a good graphical alternative.

To execute the commands contained in the file **numusers**, all you now have to do is type **./numusers** at the command line.

```
$ ./numusers
bash: ./numusers: Permission denied
$
```

So what went wrong? Before you can execute a program this way, you must change the file’s permissions and make it **executable**. Use the **chmod** command to do this:

```
$ chmod 755 numusers
```

1. So as the first part of this exercise, create and run the above shell script.

---

<sup>1</sup> Credits: This tutorial is adapted from the original version written by Willie Hayes.

You can put any commands at all inside a file, make the file executable, and then execute its contents simply by typing its name to the shell. It's that simple and that powerful.

2. As an extension to the script above, use the shell script in conjunction with the file redirection operator to redirect the output to a file called "howmany"; i.e.  
`$ ./numusers > howmany`
3. To see what has been written to the **howmany** file, enter  
`$ cat howmany`

### Example 2

For the next example suppose you want to write a shell program called "stats" that prints the date and time, the number of users logged in, and your current working directory. You know that the three command sequences you need to get this information are:

**date**, **who | wc -l**, and **pwd**.

```
$ nano stats
Enter the lines:
date
who | wc -l
pwd
$ chmod 755 stats
$ ./stats
Thu May 15 14:12:23 IST 2014
3
/home/ccahill/
$
```

To make the output of this script a bit more user friendly we can add some text to the script, to appear at certain points of the program's execution. This functionality is achieved through the use of the shell command **echo**. Add the following lines to your **stats** program (indicated using *italics*)

```
$ nano stats
echo The current time and date is:
date
echo
echo The number of users on the system is:
who | wc -l
echo
echo Your current working directory is:
pwd
$ ./stats
.....
```

What is the output of the above program, and what is the significance of the **echo** statement on a line without any preceding text?

### Comments

The Shell Programming language would not be complete without a *comment statement*. A comment statement allows you to insert remarks or comments about your program without it having any effect on its execution. Whenever the shell encounters the special character # at the beginning of a word or line, it takes whatever characters following the # as comments and simply ignores them; e.g.

```
# Author: C Cahill
# Program Description:
# This program calculates the amount of CPU usage of each user
.....
```

Comments are useful for documenting commands or sequences of commands whose purpose may not be obvious.

## **Variables**

Like virtually all programming languages, the shell allows you to store values into *variables*. To store a value in a shell variable, you simply write the name of the variable followed immediately by the equals sign =, followed immediately by the value you wish the variable to contain (**N.B. do not use spaces**)

```
variable_name=value
e.g.
$ count=1
```

To assign the value /home/username/bin to the variable mybin, you simply write

```
$ mybin=/home/username/bin      (replace username with your user name)
```

Two points are worth noting at this stage:

1. Spaces are **not** permitted on either side of the equals sign (as mentioned above)
2. Unlike most programming languages the shell has no concept of *data types*. Whenever you assign a value to a shell variable, no matter what it is the shell simply interprets that value as a string of characters. So when we stored the value 1 into the shell variable *count* above, the shell made no distinction whatsoever that an integer value was being stored in the variable.

## **Displaying the values of Variables**

Using the **echo** command encountered above, we can display the value stored in a shell variable as follows:

```
echo $variable_name
```

The shell recognises the \$ as a special character, if a valid variable name follows the \$ the shell understands that it is to substitute the value in place of the variable name.

```
e.g.  $ echo $count
      $ echo $mybin
```

## **Program input**

Using the **read** command allows us to get input from the user into a variable:

```
read variable_name          (note no $ sign before variable name)
```

```
e.g.  $ read number
      365                      (entered by user)
      $ echo $number
      365                      (output)
```

## Arithmetic

Use the **expr** command to perform calculations:

```
e.g. $ x=8
      $ y=5
      $ expr $x + $y
      13
```

(note that spaces required either side of + sign)

Take care with \* as this has another meaning in the shell – i.e.

```
$ expr $x "*" $y
```

## Read/Arithmetic Example Script

```
$ nano addition

# Take two numbers from the user and print their sum
echo Enter the first number
read num1
echo Enter the second number
read num2
echo
echo The sum is:
expr $num1 + $num2

$ chmod 755 addition
$ ./addition
```

## Decision Control

Use the **if/then/else/fi** commands.

The general format is:

```
if [ condition ]
then
    commands
else
    commands
fi
```

(note that spaces required either side of square brackets)

(else part is optional)

## Decision control Example Script

```
$ nano rmfiles

# This script removes all files in the ~/temp directory
echo "This will remove all files in the temp directory!"
echo "Are you sure (y/n)?"
read response
if [ $response = "y" ]
then
    rm ~/temp/*
    echo Files removed
```

```

else
    echo Not removed
fi

$ chmod 755 rmfiles
$ ./rmfiles

```

## More on Conditions

For comparing strings, use

=	equals	(surrounded by spaces)
!=	not equals	

For comparing integers, use:

-eq	equal to
-ne	not equal to
-lt	less than
-le	less than or equal to
-gt	greater than
-ge	greater than or equal to

**Example:** This script tells if there are more than 50 users logged on

```

numusers=`who | wc -l`
if [ $numusers -gt 50 ]
then
    echo "More than 50 users!"
fi

```

**Note:** The case instruction provides an alternative kind of decision control.

## Loops:

Programming languages have loop constructs to allow a block of code to be executed repeatedly, either a fixed number of times, or while some condition holds.

**while** instruction general format

```

while [ condition ]
do
    commands
done

```

**Example:** The following script prints all numbers from 1 to 100

```

# Prints all numbers from 1 to 100
#
num=1
while [ $num -le 100 ]
do
    echo $num
    num=`expr $num + 1`
done

```

An alternative is to use the **until** statement

e.g replace the 4<sup>th</sup> line above with **until [ \$num -gt 100 ]**

Another kind of loop is the **for** loop

General format:

```
for var in varlist
do
    commands
done
```

The **for** statement executes the commands once for each value in the list

#### for example 1

```
# prints the numbers from 101 to 105
#
for num in 1 2 3 4 5
do
    expr $num + 100
done
```

#### for example 2

```
# removes all Java source files in current directory. Use with care!
#
for filename in *.java
do
    rm $filename
done
```