Essentials for Scientific Computing: Bash Shell Scripting Day 3

Ershaad Ahamed
TUE-CMS, JNCASR

May 2012

1 Introduction

In the previous sessions, you have been using basic commands in the shell. The bash shell environment is in fact a complete programming environment. The large set of commands can be combined with programming constructs to perform powerful and useful functions. In the next sessions you will see how this can be done.

2 Script Files

A script file is a plain text file that contains commands that can be interpreted by the bash shell. Since it is a plain text file, you may edit or create it using your favourite text editor (Vim?). Conventionally bash script files are named with the extension `.sh`. In order to let the OS know that a file is in fact a bash script file, you must

1. Set the file to be executable, for instance

   ```
   chmod u+x myscript.sh
   ```

   This tells the OS that this file contains a program and is executable. The next step tells the OS how to actually execute it.

2. Set the first line of your script to be

   ```
   #!/bin/bash
   ```

   This will tell the OS to use bash to execute the commands in your file

3 Beginning to Script

3.1 Sequence of commands

The simplest form of scripting is to simply put a sequence of commands in a file. These are the same commands that you would type at the shell prompt. You

can create a simple script file `script1.sh` using any editor with the following lines in it and save it.

```
#!/bin/bash

echo "Listing of current directory"
ls
echo "Listing of root directory"
cd /
ls
```

You can then execute the script file using

```
./script1.sh
```

after making it executable with `chmod u+x script1.sh`.

You are already familiar with the ls command. The `echo` command simply writes the string which is passed to the command to stdout. The script prints out the listing of the files in the current directory, then changes the current working directory to the root directory and finally lists the files over there. You may run these commands one after the other at the shell prompt and will get the same output. But now we have conveniently grouped the commands together so that each time we need to execute the group of commands, we just need to run the script file.

Before we move on to more advanced scripting, there is a subtle but important difference between running `script1.sh` and running each of the commands individually at the prompt. You will find that after executing the last individual command at the prompt, your current working directory (which can be found using the `pwd` command) is now `/`. Whereas after the script has run, the current working directory remains unchanged. This is because, when a script file is executed, the shell starts a new session of the shell (called a *subshell*), and it is the subshell that interprets and executes the commands in the script. The subshell process terminates once the script is complete and we are returned to our original shell prompt. Since the `cd` command in the script changes the current working directory in the subshell and not the shell session we are in, it does not affect the state of our current shell.

4 Shell Plumbing

In the bash shell, the pipe or `|` symbol connects two commands together. For example the line

```
command1 | command2
```

connects the standard output (stdout) of `command1` to the standard input (stdin) of `command2`. Consider a simple example. The `sort` command sorts the lines of a text file alphabetically and writes it to stdout. The `uniq` command omits repeated lines in files and writes out the result to stdout (read their man pages to know more).

Consider the text file `fruits.txt` containing the following lines

```
apple
mango
orange
pineapple
mango
strawberry
strawberry
apple
grapes
```

We wish to remove duplicate lines in this file. Since `uniq` will only omit consecutive duplicate lines, we need to sort the file alphabetically first. We would like to take advantage of shell pipes to do this in one step. Our script to perform this `uniquefruits.sh` will look like.

```
#!/bin/bash

sort fruits.txt | uniq
```

5 Redirection

Suppose you would like the output of a command to be written to a file rather than the screen. The shell provides the redirection operator to achieve that. Redirection has several variants, but right now we will use the output redirection operator '`>`'. If we use

```
command > file.txt
```

This redirects the stdout of `command` so that its contents is written to the file `file.txt` rather than to the screen. You can modify `uniquefruits.sh` so that its output goes to the file `uniquefruits.txt`. The script will look like.

```
#!/bin/bash

sort fruits.txt | uniq > uniquefruits.txt
```

More than two commands can be joined in a pipeline. For example

```
#!/bin/bash

sort fruits.txt | uniq | nl > uniquefruits.txt
```

will include a line number for each line of the file, which is exactly what the `nl` command does.

6 Command Line Arguments

The `uniquefruits.sh` script above is quite inflexible since it requires that the name of the input file `fruits.txt` to be written in the program itself. This is bad programming practice. Remember how the `sort` command takes the name of the input file just after the command itself at the shell prompt. This is known

as passing a command line argument to the program. We would like to invoke `uniquefruits.sh` with the name of the input file as a command line argument as such

```
./uniquefruits.sh fruits.txt
```

There are special variables that are accessible within a script. These variables contain the command line arguments passed to the command. The first command line argument is in variable `$1`, the second in `$2` and so on. The UNIX convention is that the zeroth command line argument is the command name itself. Thus if our script is executed as

```
./uniquefruits.sh fruits.txt vegetables.txt
```

the variables will have the following values.

`$0`	`./uniquefruits.sh`
`$1`	`fruits.txt`
`$2`	`vegetables.txt`

Rewriting our script so that it accepts the name of the input file as a command line argument, we have, `uniquefruits2.sh`

```
#!/bin/bash

sort "$1" | uniq
```

We will talk about the quotes around `$1` in a later section.

7 Conditionals and Exit Status

Every command that is executed from the shell returns an exit status when it completes. The exit status is an integer and indicates whether the command executed successfully or encountered an error. Conventionally, the exit status of commands is `0` for success and non-zero when there was an error or the command failed.

7.1 The `grep` Command

```
grep [OPTIONS] PATTERN [FILE...]
```

`grep` searches for PATTERN within the input FILE or stdin and prints any line that matches PATTERN. For now consider that PATTERN is just a word or string. For example if we run `grep` on the file `fruits.txt` that we created earlier

```
grep apple fruits.txt
```

The output is

```
apple
pineapple
apple
```

Notice that `grep` finds a match even when PATTERN is within another string, in this case 'apple' is found within 'pineapple'. In order to match only whole words, we pass the `-w` option to grep.

```
grep -w apple fruits.txt
```

Giving the output

```
apple
apple
```

`grep`, like other commands, also returns an exit status. It returns `0` if it finds a match and non-zero if it doesn't or encounters an error. We will use the exit status of `grep` to write a very naïve script `isscript.sh` that tells us whether a file is a bash shell script or not. We will have to use a conditional expression to do this.

```
#!/bin/bash

if grep "/bin/bash" "$1"
then
  echo "File is a shell script"
else
  echo "File is not a shell script"
fi
```

If `grep` finds the string `#!/bin/bash` within the file, it returns `0`, and the lines between the `then` and the `else` are executed. Otherwise the lines after the `else` are executed. The `fi` marks the end of the `if` block or construct.

Since `grep`, by default, prints the matching line, you will notice that the above script prints the line `#!/bin/bash` before printing the message `File is a shell script`. In order to suppress this, we change the `if` line to

```
if grep "/bin/bash" "$1" > /dev/null
```

This redirects the stdout of the `grep` command to the file `/dev/null`. `/dev/null` is a UNIX *device special* file. That is, it does not really exist on disk. Any data written to this file is simply lost and is not stored anywhere.

Besides the exit status of commands, `if` can be used in conjuction with the `[ ]` construct to test a variety of conditions on files and variables (type `help test` for details). For example to test if the pathname contained in `$1` exists, use

```
if [ -e "$1" ]
```

The condition is true if the file exists. To negate the condition use

```
if [ ! -e "$1" ]
```

String comparison can also be done using,

```
if [ "$1" = "" ]
```

which is true if no command line argument was passed. Notice the spaces between the keywords ('=', '-e'), brackets and variables, these spaces are required. Other familiar operators are <, >, !=, all of which operate lexicographically. The equivalent arithmetic operators are -eq, -lt, -gt and -ne (see help test for a full listing).

Conditions can be combined by using boolean operators

```
if [ $a -eq 40 ] && [ $b -eq 51 ]
```

Which is true if both variable a equals 40 *and* variable b equals 51. If we wanted a logical *or* instead of *and*, we would use.

```
if [ $a -eq 40 ] || [ $b -eq 51 ]
```

8 Loops

Loops are used to execute a sequence of commands repeatedly either a fixed number of times or until some condition is reached.

8.1 The `while` Loop

Let us modify our uniquefruits.sh script so that it can work for more than one file specified on the command line. Like

```
./uniquefruits.sh fruits.txt vegetables.txt objects.txt
```

In order to process each of the files specified on the command line we need to enclose our previous script inside a loop. The loop will execute the code for each of the files. Let us try and use the while loop for this. Before we write a loop, we need to decide when the loop should terminate. That is some condition, which when reached, stops the loop from repeating. Since we'll assume that we do not know how many files will be specified on the command line by the user or our script, we will design the script so that it will work for any number of files.

The shift built-in command in the shell shifts all the positional parameters (that is $1, $2...) left by one. That is after the shift command, $1 will have the value that was previously in $2, $2 will have the value that was previously in $3, and so on. If a positional parameter does not exit, then it contains the empty string "". In our script we check for the empty string which tells us that we have run out of command line parameters and the loop should terminate.

```
#!/bin/bash

while [ -n "$1" ]
do
  sort "$1" | uniq
  shift
done
```

The while loop continues to execute the lines between do and done as long as the condition specified is true. Each time the loop executes the value of $1 changes since the shift command is executed. In the example above, the condition is [-n "$1"] which is true if $1 is any non-empty string and false if $1 is the empty string (see help test).

8.2 The `for` Loop

Let us rewrite the example above using the `for` loop. Before that we need to introduce the `$@` shell built-in variable. When `$@` appears in a script, it expands to all the command line arguments. For example if we run the command

```
./uniquefruits.sh fruits.txt vegetables.txt extras.txt
```

`$@` will expand to (be replaced by the shell with) `fruits.txt vegetables.txt extras.txt`. Our example using the `for` loop will look like

```
#!/bin/bash

for filename in "$@"
do
  sort "$filename" | uniq
done
```

Suppose we run the script with the command line arguments `fruits.txt vegetables.txt extras.txt`, `$@` will be expanded, and the line with the `for` will actually be interpreted by the shell as

```
for filename in fruits.txt vegetables.txt extras.txt
```

In the above script, `filename` is a variable. `filename` is assigned each of the values specified after the `in` keyword in sequence for each repetition of the loop. Whenever `$filename` is encountered in the script, the shell expands it (replaces it with) the actual value contained in it. Thus in the first repetition of the loop, `$filename` expands to `fruits.txt`, in the second repetition of the loop, it expands to `vegetables.txt`, and in the third repetition, it expands to `extras.txt`. Since there are no more items following the `in` keyword to assign to `filename`, the loop terminates.