Shell Programming II[1]

This self-paced tutorial sheet assumes knowledge of shell programming features discussed already, so please complete Part 1 first.

Some useful text processing tools

sort reorders the lines of the input

e.g.
```
$ nano cities
Dublin, Ireland
London, UK
Oslo, Norway
Lisbon, Portugal
$ sort cities
Dublin, Ireland
Lisbon, Portugal
London, UK
Oslo, Norway
```

Note: **nano** is the command-line text editor used in these examples, but you may use any text editor you wish. For example **gedit** is a good graphical editor. Other popular text editors include **vi** (or **vim**) and **emacs**.

```
$ sort cities > cities2
$ cat cities2
Dublin, Ireland
Lisbon, Portugal
London, UK
Oslo, Norway
```

(this would create a new file called "cities2" with the list sorted)

grep searches for a specific pattern of characters within its input. If the pattern is found, it displays the line containing it. The pattern is more generally known as a regular expression, which allows for wildcards, etc.

e.g.
```
$ grep on cities
London, UK
Lisbon, Portugal
```

The **-i** option of grep is useful (denotes case-insensitive)

e.g.
```
$ grep Lo cities
London, UK
```

e.g.
```
$ grep -i Lo cities
London, UK
Oslo, Norway
```

awk is an entire text-processing language, but it is also very useful for some small tasks, such as for extracting pieces of text from a line returned by *grep*.

e.g.
```
$ grep on cities | awk '{print $2}'
UK
Portugal
$ grep on cities | awk '{print $1}'
London,
Lisbon,
$ grep on cities | awk -F, '{print $1}'
London
Lisbon
```

find is a file search utility. It enables files conforming to some description to be found in the directory tree.

e.g.
```
$ find /home -name .bashrc
/home/jmcgibney/.bashrc
```

[1] *Credits: This tutorial is adapted from the original version written by Willie Hayes.*

Effective use of "quotes" in the Shell

Basically, there are four different types of quote characters that the shell recognises:

1. The single quote character **'**
2. The double quote character **"**
3. The backslash character \
4. The back quote character `

The first two characters and the last must appear in pairs to offer a practical use while programming within the shell. Each of these quote characters has a distinct meaning within the shell. To illustrate the difference between the quote characters, we will examine the outputs produced while working with our "Phone Book".

The "Phone Book" in this case, is a simple text file containing customer names and their associated phone numbers. The phone book takes the following format:

```
$ nano phonebook
Carlos Santana          086-45345099
Caroline Smith          01-2342689
Jimmy Page              051-23452334
Joe Walsh               087-567645123
Martina Blackmore       051-398475
Robert Plant            045-3560994
Robert Smith            045-4534534
```

Single Quotes

One use of quotes is to keep characters that are otherwise separated by whitespace (blanks) together. Lets look at an example using our phonebook as described above.

To look someone up in our phonebook file – which has been kept small here for demonstration purposes, we would use the `grep` command. (grep is used to search for a specified pattern in a file)

```
$ grep Joe phonebook
Joe Walsh               087-567645123
```

Look at what happens when we look up `Robert`:

```
$ grep Robert phonebook
Robert Plant            045-3560994
Robert Smith            045-4534534
```

We have two entries in our phone book for `Robert`, thus explaining the two lines of output. One way to overcome this problem would be to further qualify the name. For example we could specify the last name as well.

```
$ grep Robert Plant phonebook
grep: Plant: No such file or directory
Robert Plant            045-3560994
Robert Smith            045-4534534
```

So what's all that about?

Well, the first thing to recall is that the shell uses one or more white space characters to separate the arguments on the line, and the above command line results in `grep` being passed three arguments: `Robert`, `Plant` and `phonebook`. This is where we use our single quote characters. To prevent the shell from interpreting the `Robert` and `Plant` as two separate arguments, we can enclose them in single quotes. In this fashion the shell will interpret the string "Robert Plant" as a single argument while processing the `grep` command. So if we restructure our command as follows,

```
$ grep 'Robert Plant' phonebook
Robert Plant            045-3560994
```

we will retrieve the correct entry. In this case the shell encountered the first ‘, and ignored all special characters (i.e. the whitespace) which followed, until it encountered the closing ‘, thus treating all the text enclosed in the single quotes as a single argument. *grep* then took this argument (minus the quotes i.e. Robert Plant) and performed the usual *grep* search.

Double Quotes

Double quotes work similar to single quotes, except they are not quite so constrictive. Using single quotes we are telling the shell to ignore all enclosed text (with the quotes), whereas when we use double quotes we are telling the shell to ignore most characters. In particular the following three characters are not ignored by the shell when enclosed in double quotes

1. Dollar sign **$**
2. Back quote `
3. Backslash \

The fact that dollar signs are not ignored means that variable substitution is done by the shell in double quotes.

1.
```
$ x='Robert Plant'
$ echo '$x'
$x
```

2.
```
$ echo "$x"
Robert Plant
```

3.
```
$ grep "$x" phonebook
Robert Plant            045-3560994
```

The Backslash

Basically, the back slash is equivalent to placing single quotes around a single character, with a few minor exceptions. The backslash quotes the single character that immediately follows it. The general format is:

```
\c
```

where `c` is the character you want to quote. Any special meaning normally attached to that character is removed. Here are two examples:

1.
```
$ echo >
bash: syntax error near unexpected token `newline'
```

2.
```
$ echo \>
>
$
```

In the first case (1), the shell saw the > and thought you wanted to redirect `echo`’s output to a file, so it was expecting a file name to follow. When it didn’t find one, it issued the error message. In the second instance, the \ removed the special meaning of the >, and simply passed the > character to `echo` to be displayed on screen.

> What is the output of executing the following command?
> ```
> $ echo "\$x"
> ```
> Can you explain why?

Back Quotes

The back quote is unlike any of the previously encountered quote characters. Its purpose is not to protect characters from the shell but to tell the shell to execute the enclosed command and to insert the output from the point on the command line. For example:

```
$ echo The date and time is: `date`
The date and time is: Thu Apr 24 18:50:52 IST 2014
```

When the shell does its initial scan of the command line, it notices the back quote and expects to the name of a command to follow. In this case, it finds the `date` command. So it executes `date` and replaces `` `date` `` on the command line with the output of `date`.

Command Line Arguments

Shell programs become much more useful when you learn how to process arguments passed to them via the command line.

Up to this point, every time we have executed a shell program, the shell has automatically stored all the proceeding arguments in what is referred to as *positional parameters*, namely 1, 2, 3, etc. These parameters (arguments) can be referenced using the $ sign. For example if we had a shell program called *findproc,* which searches for processes running on the system (which we would specify on the command line), it could take the following format:

```
$ ./findproc bash
```

In this case, the shell assigns `$1` to the string `bash`. In this form, we can then do the appropriate search on the logged-on users, performing a `grep` on `$1`.

Okay, let's create the above mentioned shell script.

1. First of all create the file `findproc`
   ```
   $ nano findproc
   ps -ef | grep $1
   ```
   (save and exit; also make executable with `chmod +x findproc`)

2. Execute the script passing to it your user name
   ```
   $ ./findproc bash
   ```
   Notice how the shell substitutes the value denoted by `$1` and allows us to perform the `grep`.

The $# Variable

Whenever you execute a shell program, the special shell variable `$#` is set to ***the number of arguments*** that were passed to the script on the command line. This variable can then be tested by a program to determine if the user typed the correct number of arguments.

Examine the output of the following shell script (`args`); it will help you get more familiar with the way in which arguments can be passed to a script.

1. Create the shell program.
   ```
   $ nano args
   echo $# arguments passed
   echo arg 1 = $1    arg 2 = $2    arg 3 = $3
   ```
   (save, exit and make executable)
2. Execute it
   ```
   $ ./args a b c
   3 arguments passed
   arg 1 = a    arg 2 = b    arg 3 = c
   ```
3. Try it with 2 arguments
   ```
   $ ./args a b
   2 arguments passed
   arg 1 = a    arg 2 = b    arg 3 =
   ```
4. Try it with 0 arguments
   ```
   $ ./args
   0 arguments passed
   arg 1 =     arg 2 =     arg 3 =
   ```
5. Try it with double quotes
   ```
   $ ./args "a b c"
   1 arguments passed
   arg 1 = a b c   arg 2 =     arg 3 =
   ```

As you can see the shell does its normal command line processing, even when it's executing your shell program. This means that you can take advantage of the normal niceties like filename substitution and variable substitution when specifying arguments to your programs.

`$#` is often used to check that the user has entered the correct number of command line arguments.

Example:

```
$ nano addnums
# Sums two numbers supplied on the command line
#
if [ $# -ne 2 ]
then
    echo Please provide two command line arguments
    echo Correct usage:
    echo $0 num1 num2
else
    total=`expr $1 + $2`
    echo The sum is $total
fi
```

Usage:

```
$ ./addnums 7 16
The sum is 23
```

The $* Variable

The special variable `$*` references ***all the arguments*** passed to a program. This is often useful in programs that take an indefinite or variable number of arguments. Therefore we could modify our args program as follows, which would now echo all the arguments passed to it, to the console.

```
$ nano args
echo $# arguments passed
echo they are $*
```

(save, exit and make executable)

```
$ ./args a b c d e f
echo 6 arguments passed
echo they are a b c d e f
```

So, keeping what we have just learned in mind, let's go back to our phone book example and take a quick look at its contents:

```
$ cat phonebook
Carlos Santana          086-45345099
Caroline Smith          01-2342689
Jimmy Page              051-23452334
Joe Walsh               087-567645123
Martina Blackmore       051-398475
Robert Plant            045-3560994
Robert Smith            045-4534534
```

Up until now, we have simply searched our phone book for specific entries, i.e.:

```
$ grep 'Robert Plant' phonebook
Robert Plant            045-3560994
```

Wouldn't it be nice if we could develop some scripts that would allow us to update our phonebook (i.e. add and remove existing entries) without having to always manually edit the file?

In this final look at command line arguments we are going to develop three new scripts. The first script (called *phonelu – short for phone look up*) will be an argument-based version of the search utility we have already used on our phone book. The second script (called *phoneadd*) will allow us to add a new entry to our phone book and, finally, our third script (*phonerm*) will allow us to remove an existing entry from our phone book based on a specified user name.

1. *phonelu*

```
$ nano phonelu
grep  "$1" phonebook
```
(save, exit and make executable)

Now let's try it:
```
$ ./phonelu Caroline
Caroline Smith          01-2342689
$ ./phonelu "Joe W"
Joe Walsh               087-567645123
```

Q. What's the significance of having the `$1` in double quotes?

2. *phoneadd*

```
$ nano phoneadd
echo  "$1   $2"  >>  phonebook
```
(save, exit and make executable)
(the first & second arguments are separated by a tab space)

Now lets try it:
```
$ ./phoneadd 'Don Henley'  051-23487347
$ ./phonelu Don
Don Henley        051-23487347
```

This program takes 2 command line arguments

```
$ cat phonebook
Carlos Santana          086-45345099
Caroline Smith          01-2342689
Jimmy Page              051-23452334
Joe Walsh               087-567645123
Martina Blackmore       051-398475
Robert Plant            045-3560994
Robert Smith            045-4534534
Don Henley              051-23487347
```

In the above example *Don Henley* was quoted, so that the shell would pass it along to *phoneadd* as a single argument (what would have happened if we hadn't done this?). After *phoneadd* finished, we executed *phonelu,* to see if we could find the new entry. We then called `cat`, to list the contents of the file. So everything worked fine!

Well the file is no longer sorted (in alphabetical order), *phoneadd* only added the entry onto the end of the file. We can overcome this inconvenience by adding the following line to our script: (on a new line below the echo command)

```
sort –o phonebook phonebook
```

The `–o` option to `sort` specifies where the sorted output is to be written, and this can be the same as the input file.

3. *phonerm*

```
$ nano phonerm
grep –v  "$1"   phonebook    >  /tmp/phonebook
mv /tmp/phonebook phonebook
```
(save, exit and make executable)

Now let's try it:

```
$ ./phonerm Robert
$ cat phonebook
Carlos Santana          086-45345099
Caroline Smith          01-2342689
Jimmy Page              051-23452334
Joe Walsh               087-567645123
Martina Blackmore       051-398475
Don Henley              051-23487347
```

As an exercise, use the `man` pages to find out exactly what happens when we execute the *phonerm* script.

Functions

The shell allows you to define functions which can then be used as shell commands. Functions help to improve modularity and avoid repetition of tasks.

General format:

```
function function_name
{
      commands
}
```

Example:

```
$ nano printtext
function highlight
{
      echo "====== $* ======"
}
highlight Heading Text
echo Normal Text
highlight End of Text
```

The output of this script is:

```
$ ./printtext
====== Heading Text ======
Normal Text
====== End of Text ======
```

Exit and Return Codes

A shell script terminates at the end of the instructions, or if it reaches an exit command. The exit command can be used with an argument, and this argument can be interpreted by later programs. The default exit code is 0. The exit code from a previous command can be accessed as $?

Example:

```
$ nano yearsago
# Returns the number of years ago a certain year was
#
if [ $1 -gt 2014 ]
then
        exit 1
else
        echo That was `expr 2014 - $1` years ago
fi
```

Usage:

```
$ ./yearsago 1969
That was 45 years ago
$ echo $?
0
$ ./yearsago 2020
$ echo $?
1
```

At any time, you can check the exit code returned by a command including the standard commands;

e.g.

```
$ ls
hello.java   goodbye.java
$ echo $?
0
$ ls xyz*
ls: xyz*: No such file or directory
$ echo $?
1
```

Functions within scripts may also return a value.

Exercise:

The `yearsago` script has the current year hard-coded. Modify the script so it will work next year or any year. *Hint: have a look at the* **`man`** *page for the* **`date`** *command.*