# A quick guide to writing scripts using the bash shell

## A simple shell script

A shell script is little more than a list of commands that are run in sequence. Conventionally, a shellscript should start with a line such as the following:

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

THis indicates that the script should be run in the bash shell regardless of which interactive shell the user has chosen. This is very important, since the syntax of different shells can vary greatly.

### A simple example

Here's a very simple example of a shell script. It just runs a few simple commands

```
#!/bin/bash
echo "hello, $USER. I wish to list some files of yours"
echo "listing files in the current directory, $PWD"
ls  # list files
```

Firstly, notice the comment on line 4. In a bash script, anything following a pound sign # (besides the shell name on the first line) is treated as a comment. ie the shell ignores it. It is there for the benifit of people reading the script.

$USER and $PWD are *variables*. These are standard variables defined by the bash shell itself, they needn't be defined in the script. Note that the variables are *expanded* when the variable name is inside double quotes. Expanded is a very appropriate word: the shell basically sees the string $USER and replaces it with the variable's value then executes the command.

We continue the discussion on variables below ...

## Variables

Any programming language needs variables. You define a variable as follows:

```
X="hello"
```

and refer to it as follows:

```
$X
```

More specifically, $X is used to denote the value of the variable X. Some things to take note of regarding semantics:

- bash gets unhappy if you leave a space on either side of the = sign. For example, the following gives an error message:

  ```
  X = hello
  ```

- while I have quotes in my example, they are not always necessary. where you need quotes is when your variable names include spaces. For example,

```
X=hello world # error
X="hello world" # OK
```

This is because the shell essentially sees the command line as a pile of commands and command arguments seperated by spaces. `foo=bar`is considered a command. The problem with `foo = bar` is the shell sees the word `foo` seperated by spaces and interprets it as a command. Likewise, the problem with the command `X=hello world` is that the shell interprets `X=hello` as a command, and the word "world" does not make any sense (since the assignment command doesn't take arguments).

**Single Quotes versus double quotes**

Basically, variable names are exapnded within double quotes, but not single quotes. If you do not need to refer to variables, single quotes are good to use as the results are more predictable.

An example

```
#!/bin/bash
echo -n '$USER=' # -n option stops echo from breaking the line
echo "$USER"
echo "\$USER=$USER"  # this does the same thing as the first two lines
```

The output looks like this (assuming your username is elflord)

```
$USER=elflord

$USER=elflord
```

so the double quotes still have a work around. Double quotes are more flexible, but less predictable. Given the choice between single quotes and double quotes, use single quotes.

**Using Quotes to enclose your variables**

Sometimes, it is a good idea to protect variable names in double quotes. This is usually the most important if your variables value either (a) contains spaces or (b) is the empty string. An example is as follows:

```
#!/bin/bash
X=""
if [ -n $X ]; then      # -n tests to see if the argument is non empty
        echo "the variable X is not the empty string"
fi
```

This script will give the following output:

```
the variable X is not the empty string
```

Why ? because the shell expands $X to the empty string. The expression [ -n ] returns true (since it is not provided with an argument). A better script would have been:

```
#!/bin/bash
X=""
if [ -n "$X" ]; then    # -n tests to see if the argument is non empty
```

```
        echo "the variable X is not the empty string"
fi
```

In this example, the expression expands to [ -n "" ] which returns false, since the string enclosed in inverted commas is clearly empty.

### Variable Expansion in action

Just to convince you that the shell really does "expand" variables in the sense I mentioned before, here is an example:

```
#!/bin/bash
LS="ls"
LS_FLAGS="-al"

$LS $LS_FLAGS $HOME
```

This looks a little enigmatic. What happens with the last line is that it actually executes the command

```
ls -al /home/elflord
```

(assuming that /home/elflord is your home directory). That is, the shell simply replaces the variables with their values, and then executes the command.

### Using Braces to Protect Your Variables

OK. Here's a potential problem situation. Suppose you want to echo the value of the variable X, followed immediately by the letters "abc". Question: how do you do this ? Let's have a try :

```
#!/bin/bash
X=ABC
echo "$Xabc"
```

THis gives no output. What went wrong ? The answer is that the shell thought that we were asking for the variable Xabc, which is uninitialised. The way to deal with this is to put braces around X to seperate it from the other characters. The following gives the desired result:

```
#!/bin/bash
X=ABC
echo "${X}abc"
```

## Conditionals, if/then/elif

Sometimes, it's necessary to check for certain conditions. Does a string have 0 length ? does the file "foo" exist, and is it a symbolic link , or a real file ? Firstly, we use the if command to run a test. The syntax is as follows:

```
if condition
then
        statement1
        statement2
        ..........
fi
```

Sometimes, you may wish to specify an alternate action when the condition fails. Here's how it's done.

```
if condition
then
        statement1
        statement2
        ..........
else
        statement3
fi
```

alternatively, it is possible to test for another condition if the first "if" fails. Note that any number of elifs can be added.

```
if condition1
then
        statement1
        statement2
        ..........
elif condition2
then
        statement3
        statement4
        ........
elif condition3
then
        statement5
        statement6
        ........


fi
```

The statements inside the block between `if/elif` and the next `elif` or `fi` are executed if the corresponding condition is true. Actually, any command can go in place of the conditions, and the block will be executed if and only if the command returns an exit status of 0 (in other words, if the command exits "succesfully" ). However, in the course of this document, we will be only interested in using "test" or "[ ]" to evaluate conditions.

### The Test Command and Operators

The command used in conditionals nearly all the time is the test command. Test returns true or false (more accurately, exits with 0 or non zero status) depending respectively on whether the test is passed or failed. It works like this:

```
test operand1 operator operand2
```

for some tests, there need be only one operand (operand2) The test command is typically abbreviated in this form:

```
[ operand1 operator operand2 ]
```

To bring this discussion back down to earth, we give a few examples:

```
#!/bin/bash
X=3
```

```
Y=4
empty_string=""
if [ $X -lt $Y ]        # is $X less than $Y ?
then
        echo "\$X=${X}, which is greater than \$Y=${Y}"
fi

if [ -n "$empty_string" ]; then
        echo "empty string is non_empty"
fi

if [ -e "${HOME}/.fvwmrc" ]; then                      # test to see if ~/.fvwmrc exists
        echo "you have a .fvwmrc file"
        if [ -L "${HOME}/.fvwmrc" ]; then              # is it a symlink ?
                echo "it's a symbolic link
        elif [ -f "${HOME}/.fvwmrc" ]; then     # is it a regular file ?
                echo "it's a regular file"
        fi
else
        echo "you have no .fvwmrc file"
fi
```

## Some pitfalls to be wary of

The test command needs to be in the form "operand1<space>operator<space>operand2" or operator<space>operand2 , in other words you really *need* these spaces, since the shell considers the first block containing no spaces to be either an operator (if it begins with a '-') or an operand (if it doesn't). So for example; this

```
if [ 1=2 ]; then
        echo "hello"
fi
```

gives exactly the "wrong" output (ie it echos "hello", since it sees an operand but no operator.)

Another potential trap comes from not protecting variables in quotes. We have already given an example as to why you *must* wrap anything you wish to use for a `-n` test with quotes. However, there are a lot of good reasons for using quotes all the time, or almost all of the time. Failing to do this when you have variables expanded inside tests can result in *very* wierd bugs. Here's an example: For example,

```
#!/bin/bash
X="-n"
Y=""
if [ $X = $Y ] ; then
        echo "X=Y"
fi
```

This will give misleading output since the shell expands our expression to

```
[ -n = ]
```

and the string "=" has non zero length.

## A brief summary of test operators

Here's a quick list of test operators. It's by no means comprehensive, but its likely to be all you'll need to remember (if you need anything else, you can always check the bash manpage ... )

| operator | produces true if... | number of operands |
|---|---|---|
| -n | operand non zero length | 1 |
| -z | operand has zero length | 1 |
| -d | there exists a directory whose name is *operand* | 1 |
| -f | there exists a file whose name is *operand* | 1 |
| -eq | the operands are integers and they are equal | 2 |
| -neq | the opposite of -eq | 2 |
| = | the operands are equal (as strings) | 2 |
| != | opposite of = | 2 |
| -lt | *operand1* is strictly less than *operand2* (both operands should be integers) | 2 |
| -gt | *operand1* is strictly greater than *operand2* (both operands should be integers) | 2 |
| -ge | *operand1* is greater than or equal to *operand2* (both operands should be integers) | 2 |
| -le | *operand1* is less than or equal to *operand2* (both operands should be integers) | 2 |

## Loops

Loops are constructions that enable one to reiterate a procedure or perform the same procedure on several different items. There are the following kinds of loops available in bash

- for loops
- while loops

### For loops

The syntax for the for loops is best demonstrated by example.

```
#!/bin/bash
for X in red green blue
do
        echo $X
done
```

THe for loop iterates the loop over the space seperated items. Note that if some of the items have embedded spaces, you need to protect them with quotes. Here's an example:

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

```
colour1="red"
colour2="light blue"
colour3="dark green"
for X in "$colour1" $colour2" $colour3"
do
        echo $X
done
```

Can you guess what would happen if we left out the quotes in the for statement ? This indicates that variable names should be protected with quotes unless you are pretty sure that they do not contain any spaces.

**Globbing in for loops**

The shell expands a string containing a * to all filenames that "match". A filename matches if and only if it is identical to the match string after replacing the stars * with arbitrary strings. For example, the character "*" by itself expands to a space seperated list of all files in the working directory (excluding those that start with a dot "." ) So

```
echo *
```

lists all the files and directories in the current directory.

```
echo *.jpg
```

lists all the jpeg files.

```
echo ${HOME}/public_html/*.jpg
```

lists all jpeg files in your public_html directory.

As it happens, this turns out to be very useful for performing operations on the files in a directory, especially used in conjunction with a for loop. For example:

```
#!/bin/bash
for X in *.html
do
                grep -L '<UL>' "$X"
done
```

**While Loops**

While loops iterate "while" a given condition is true. An example of this:

```
#!/bin/bash
X=0
while [ $X -le 20 ]
do
        echo $X
        X=$((X+1))
done
```

This raises a natural question: why doesn't bash allow the C like for loops

for (X=1,X<10; X++)

As it happens, this is discouraged for a reason: bash is an interpreted language, and a rather slow one for that matter. For this reason, heavy iteration is discouraged.

## Command Substitution

Command Substitution is a very handy feature of the bash shell. It enables you to take the output of a command and treat it as though it was written on the command line. For example, if you want to set the variable X to the output of a command, the way you do this is via command substitution.

There are two means of command substitution: brace expansion and backtick expansion.

Brace expansion workls as follows: `$(`*`commands`*`)` expands to the output of *commands* This permits nesting, so *commands* can include brace expansions

Backtick expansion expands `` ` ``*`commands`*`` ` `` to the output of *commands*

An example is given;:

```
#!/bin/bash
files="$(ls )"
web_files=`ls public_html`
echo $files
echo $web_files
X=`expr 3 \* 2 + 4` # expr evaluate arithmatic expressions. man expr for details.
echo $X
```

Note that even though the output of ls contains newlines, the variables do not. Bash variables can not contain newline characters (which is a pain in the butt. But that's life) Anyway, the advantage of the $() substitution method is almost self evident: it is very easy to nest. It is supported by most of the bourne shell varients (the POSIX shell or better is OK). However, the backtick substitution is slightly more readable, and is supported by even the most basic shells (any #!/bin/sh version is just fine)