Table of Contents

Tutorial Overview

This tutorial is at a beginner level, for new users of Linux systems that wish to become productive quickly with BASH shell scripting.

Despite the uptake of languages such as Python becoming more popular it is important to note that languages such as Python are programming languages and require additional packages installed; they are not scripting languages common on all servers, and are far to complicated to use for writing simple automation tasks or day-to-day utilities.

*Script languages are defined as those specifically provided by the interactive shell of choice for users of any *nix system*; *they are provided with the system so require no additional packages to be in installed*, and can use commands users are already familiar with without having to learn programming interfaces in higher level languages or install additional packages.

While it may seem old-hat I believe it is still best to perform all system scripting in standard shell functions, and if there is a need for a program use C.

It should also be noted that any *nix system administrator can easily maintain shell scripts written by those who have gone before for system admin and maintenance activities, while scripts written in such things as Python and PERL can be a real headache and generally require a complete replacement/rewrite by a unix system administrator to a shell script in the cases they are encountered and need updating by the unlucky system administrator that encounters them.

Also while not as ‘type-safe’ as full programming languages it is not hard to write complex webserver served applications using nothing but the shell scripting language; and while there are special considerations on parsing user input and preventing buffer overflows which I have covered in a completely separate tutorial available from my site on ‘writing web server applications using shell script’. Although you should probably review this tutorial before looking at that one.

This tutorial however is specifically for the BASH shell. The BASH shell contains variable functions that are not all available on other shells (although in swings and roundabouts other shells have facilities not available in bash; I just prefer bash).

This tutorial provides examples of the main operations and syntax you are likely to use in your BASH scripts. It is not a programming tutorial, in that you are expected to have enough basic programming knowledge in order to fit the pieces covered here together yourself.

Using Comments

The ‘bash’ shell comment character is the hash (#) character, this may be used anywhere on a line and results in all text on that line after the # character being treated as non-executable comment data.

There will be many examples of using comments in the examples I show throughout this document which will be the best way of seeing how these are used.

Multi-line Commands

For large commands which need to span multiple lines the continuation character backslash (\ ) is used. It is important to note that this needs to be at the end of commands and not inserted in the middle of a command to avoid unexpected results.

An example of use is shown in the example for the below section on using pipes.

If using multi-line commands with this continuation character is is also important not to use the comment character anywhere within that command or you will break the command; comments can be added at the end of a command but not in the middle of one as everything after the comment is treated as a comment which would be all the remaining continuation lines.

Redirecting Output

The <, >, and >> operations

Most UNIX utilities write output to the two standard file descriptors STDOUT and STDERR, although many simply only use STDOUT. **These can be referenced on the command line as 1 and 2 respectively** In places where redirection is allowed.

Redirection is performed using the left and right braces (the < and > characters), which in the simplest terms are

- < redirects the contents of a file to the STDIN of the program on the left within the command line
- << as above but the file is tailed so command appended to the file after the program starts are also executed as new commands. *This should be avoided* but can be seen in some badly written applications such as the ibm domino server that on unix servers expects its interactive commands this way (if not running in the foreground with a dedicated terminal session providing the interactive command console)
- > redirects output to the file on the right of the command with *overwrite*
- >> redirects output to the file on the right of the command with *append*.

Very seldom would you see the < redirection used; and it should be avoided. It can occasionally be seen in scripts to load/stack commands into a program but using pipes is more readable (although with more overhead than using <).

The 2>/dev/null redirection usage

The redirection of STDERR using 2>/dev/null is fairly common in shell scripts to suppress expected errors written to STDERR as

- if left undirected lots of expected non-fatal errors are shown to users of shell script
- only STDOUT is passed through pipes so errors can be discarded without impact

An example of a program that uses both output streams is the standard UNIX 'find' command, which writes errors such as permission denied on directories to STDERR and normal results to STDOUT.

To suppress STDERR from the find command you would redirect STDERR as in the command below which redirects ll error messages to /dev/null.

```
find / -type f -name “device.map” 2>/dev/null
```

The 2>&1 redirection usage

If you wish to actually merge the STDERR output with the STDOUT output the special syntax of ' 2>&1 ' is used to redirect STDOUT to STDERR. The & character is required.

So if you for some strange reason you want your data interspersed with error messages the above find command can be changed to the command below.

```
find / -type f -name “device.map” 2>&1
```

The back-quote/back-tick redirection

An additional way of redirecting output is covered in the section on “placing command output into a variable”, where the STDOUT from a command can be redirected directly into a variable using the back-quote syntax variablename=`unix command`.

Refer to that section for details.

Using Pipes

The use of UNIX pipes is not specific to bash, it is used on all UNIX systems and you should be familiar with using pipes already.

Pipes are used to pass data to subsequent programs in the pipe chain for processing; passed using the pipe character vertical bar (|).

UNIX at its core is a collection of small utilities to provide specific small discrete functions.

Pipes are important to UNIX servers as they permit chaining those discrete commands to allow data to be manipulated by those commands; data passed from a programs STDOUT through the pipe is sent to the STDIN of the process the data is being piped to. This is simply more efficient than running each command separately and having to store the output between steps.

The important thing to note when using pipes within shell scripts is that the return code from the command will be the return code from the last program or script executed, return codes from other commands in the pipe chain are not available; as such there is no easy way from within a script to determine if all the commands within a pipe command chain completed correctly.

A simple example is shown below. It pipes text to the tr utility which upper cases it, that upper case text is passed to the awk utility to print field number two.

```
echo "lower case text" \
   | tr '[:lower:]' '[:upper:]' \
   | awk {'print $2'}
```

Pipes are often used in shell scripts as apart from being more efficient it can make the code cleaner and easier to read.

Using Command Lists

This covers the || and && operators only, as they are the only common operators you will see or need to use in shell scripts.

These operators are used to control execution of commands where a command should be executed or not executed depending upon the exit code of a preceding program or script.

The basic syntax is

```
command1 && command2 #command2 executed only if command1 exit code was 0
command1 || command2 # command2 executed only if command1 exit code non-0
```

These are fairly self explanatory, are commonly seen, and can be considered safe to use.

There are other very seldom used options available, you should refer to the ‘man bash’ command if you are interested in all the options; *but you should avoid using infrequently used syntax and functions in any of your scripts as the next person tasked with maintaining the script won’t know what you were trying to do*.

Command Aliases

Aliases are mentioned as they should not be used in shell scripts, but you may inadvertently end up using some. This section is not on how to create them but how to avoid them in scripts.

Aliases are defined at a global system level affecting all users, with user specific aliases defined in users profiles. A common alias found in most sites is something like “ alias rm=’rm -i’ “ to ensure users are prompted before a file is deleted; obviously if you want to use the ‘rm’ command in a script this is going to cause a problem.

This section is to specifically highlight that in scripts never use commands such as “rm filename”, always use the full command path such as “/bin/rm filename” to bypass any aliases your site may have defined.

Note: the command “alias” with no parameters will display all the aliases you have defined.

Simple Variables

Overview of Variables

It should be noted that by default all variables are treated as text variables unless explicitly defined otherwise; this can be seen in the variable assignment section below.

Variables being text by default is not an issue as in most cases variable expansion occurs in operations, for example if a is the text value 10 then “10 + $a” will be 20 as the variable is expanded and as such can be used as a number as well as text.

It does however affect special variable modifiers which are type specific, refer to the section on setting specific types for variables for more information.

Variables also cannot be named the same as any bash reserved names such as declare, typeset, function etc. That should not be an issue as you can use meaningful long variable names such as this_is_my_variable and avoid using short names that may match a command.

Variable names also cannot begin with a $ which is reserved for accessing the variable; but as a general rule always have a character between a-z (or A-Z) as the first character of any variable name.

Simple Variable Assignment

For those used to programming languages it is important to note that in shell scripts there must be no spaces around the equals sign when performing variable assignment in bash. *Failure to adhere to that will treat the variable name as a command as shown below. This is also a good reason to ensure variable names do not match any linux command.*

Variables do not need to be pre-defined, and can be created as needed by a simple value assignment, there are very little rules in the naming of variables however ***when assigning to a variable it is important to note the variable name must NOT be preceded by the dollar ($) sign***, that is reserved for accessing the variable data.

The assignment command '=' can if you wish to append data to a simple variable can also be

```
[mark@phoenix]$ bb=ddddd   # valid
[mark@phoenix]$ bb = ddddd # invalid, spaces will treat variable as a
command
bash: bb: command not found…
[mark@phoenix]$ bb=100     # valid
[mark@phoenix]$ echo ${bb}
100
[mark@phoenix]$ bb=a test  # invalid, space and no quotes discards bad data
[mark@phoenix]$ echo ${bb} # so the echo is a blank line

[mark@phoenix]$ bb="a test"  # valid, spaces can be included within quotes
[mark@phoenix]$ echo ${bb}
a test
[mark@phoenix]$ bb=dddd      # valid (no spaces in string)
[mark@phoenix]$ echo ${bb}
dddd
[mark@phoenix]$
```

And as seen to assign a value to a variable is simply a case of 'variablename="variable value"'. Both text and numbers can by default be set for any variable as long as the following rules are followed, as demonstrated by the examples above.

- There must be no spaces around the = in the assignment or the variable is treated as a command
- If assigning a string value if the value contains spaces it must be within quotes, assigning a text value with spaces that is not surrounded with quotes will have the data discarded and the value set to empty

The assignment command ‘=’ can if you wish to append data to a simple text variable or alter the contents of a simple numeric variable can be replaced with the “+=’ syntax which has the following results

- for a text variable appends data to the existing data to the variables
- for a numeric variable it is treated as an arithmetic addition

It is however important to note that all variables by default are treated as text variables unless explicitly defined as integer, as seen in the examples below.
For how the += operator affects arrays see the array variables section.

```
[mark@phoenix]$ aa="test data"
[mark@phoenix]$ echo $aa
test data
[mark@phoenix]$ aa+=" line"     # for a string appends extra text
[mark@phoenix]$ echo $aa
test data line
[mark@phoenix]$ aa=10           # default type is string
[mark@phoenix]$ echo $aa
10
[mark@phoenix]$ aa+=10
[mark@phoenix]$ echo $aa        # so the 10 is appended
1010
[mark@phoenix]$ unset aa        # but if
[mark@phoenix]$ typeset -i aa   #   aa is declared as an integer
[mark@phoenix]$ aa=10
[mark@phoenix]$ echo $aa
10
[mark@phoenix]$ aa+=10          # the 10 is added
[mark@phoenix]$ echo $aa
20
[mark@phoenix]$
```

An additional way of placing data into a variable is covered in the section on “placing command output into a variable”.

If Uninitialised Variable Assignment (you have a major script error)

This is for use within scripts when accessing the variable, and not for setting an initial value.
If within a script you access variable ${myvar} and it is not set you would get unexpected results, there is a variable syntax access option ${myvar:=avalue} which will during variable expansion if the variable is unset will set the variable value to the provided 'avalue', if the variable was already set the existing value would be used.

If you ever have to use this syntax in your script **you have a major problem with the script**, that must be fixed, no variable used should ever be uninitialised at the time it is needed.

Should this syntax ever be used by you in any shell script you should consider yourself fired as you are obviously totally incompetent and should never be permitted to login to a computer, tablet or cellphone. If this syntax was ever found by corporate auditors in any shell script people would be fired.

It is mentioned in this document simply because you may see this syntax mentioned on websites like stackexchange in answer to users questions; because some responders like to see idiots digging themselves into a hole.

There are similar variable check/assign-value options in 'bash' variable expansion you can find in the 'bash' man page; the same warning applies, if you use them you should be fired, as it implies that you are totally incapable of initialising variables before there is an access attempt, which is incompetent coding.

But operations like these do exist, you will seem them used in badly written scripts and in answers on some forums, but use of them clearly implies you have an known error in your code you cannot be bothered to track down and fix; effectively making your code 'untrusted'.

Accessing Simple Variable Data

To reference the value of a 'bash' variable is as simple as prefixing the variable name with a dollar ($) sign. However the below paragraph is important.

While optional for *simple variables* it is also good practise when referring to the contents of a variable to surround the variable name with {} both for documentation that you know it is a variable and a good habit to get into as you need those for advanced variable manipulation functions plus for accessing non-simple variables such as arrays.

So for simple variable myvar both $myvar and ${myvar} can be used to access the variable contents. For more advanced variables you must use ${myvar}; so get into the habit of using the {} syntax to avoid issues later on.

It is important to ensure you only refer to the variable within a valid command or assignment to prevent unexpected results; such as it being treated as a command as seen below.

For Example:

```
[mark@phoenix]$ aa="variabledata"  # set variable
[mark@phoenix]$ echo "$aa"         # echo variable
variabledata
[mark@phoenix]$ echo "${aa}"     # echo variable
variabledata
[mark@phoenix]$ $aa           # oops, expanded and treated as a command
bash: variabledata: command not found...
[mark@phoenix]$
```

This of course does not mean that you should avoid using variables as commands, it is common to assemble complicated commands within shell script variables and issue the command using the variable, you just need to be very aware that if it occurs unintentionally it can have serious adverse affects.

Array Variables

Important Notes

As mentioned in the “Simple Variable Assignment” section the += assignment has unique properties explicitly associated with arrays that do not apply to normal variables.

The ‘+=’ operator if used against an array variable it will have the following results

- for an indexed array a new entry is appended to the array variable at var[max+1]
- for associative arrays a new key-value pair is added to the array variable

It is also important to note that accessing the contents of arrays can only be done using the {} syntax.

Indexed Arrays

An indexed array is an array of elements referenced by the element number.

Care needs to be taken when using indexed arrays, there is nothing to stop you creating entry 0,1,2 and then jumping directly to creating entry 100. Nothing wrong with that as referring to non-initialised entries in the empty gaps will just return null, but unless there is a good reason for it it indicates bad coding.

You can create an indexed array with lazy coding by simply assigning an element to it such as “myindexarray[0]=”line 0;myindexarray[1]=”line 1”;...” however correct coding would be something like “typeset -a myindexarray;myindexarray[0]=”line 0”;myindexarray[1]=”line1”;….”. Reguardless of which method is used an index array is created.

- If you try to update or get data from that array without indexes only entry [0] is updated or returned as seen below.
- If you want to access (read) any array entry using the index as mentioned in the earlier section on variables you must use the {} notation to do so as seen below.

```
[mark@phoenix]$ myindexarray[0]="line 0"
[mark@phoenix]$ myindexarray[1]="line 1"
[mark@phoenix]$ typeset -p myindexarray      # show variable
declare -a myindexarray=([0]="line 0" [1]="line 1")
[mark@phoenix]$ myindexarray="oops, trashed" # update with no index
[mark@phoenix]$ typeset -p myindexarray      # shows entry 0 trashed
declare -a myindexarray=([0]="oops, trashed" [1]="line 1")
[mark@phoenix]$ echo $myindexarray # treated as a normal entry 0 is used
oops, trashed
[mark@phoenix]$ echo $myindexarray[1] # cannot get element 1 without {}
oops, trashed[1]
[mark@phoenix]$ echo ${myindexarray[1]} # using {} is correct way
line 1
[mark@phoenix]$
```

As you can see from the “typeset -p” output in the example above an indexed array is actually an associative array, with numbers being the array “key”, with the exception that non-numeric keys or indexes are not permitted in indexed arrays; trying to use a non-numeric key will always result in entry [0] being used as seen in the first example in the associated arrays section below.

For indexed arrays the ‘bash’ built in commands ‘mapfile’ and ‘readarray’ (are the same function, but the backward compatible name is still supported) can be used to read lines from STDIN (or a file descriptor) and load them into an indexed array variable.

In its simplest form the syntax to load a small file into an indexed array variable named arrayvar would be

```
cat smallfilename | readarray arrayvar
```

The mapfile|readarray built in functions have a lot of options, use ‘man bash’ to see what options are available (such as discarding lines from the STDIN, and appending to an array instead of re-initialising it).

It is also possible to initialise an indexed array from a list with the syntax shown below. The below syntax may be seen occasionally in scripts for word number extraction from a string, but it is more common to use “awk” for word extraction for portability.

```
N=3
STRING="one two three four"
arr=($STRING)
echo ${arr[N-1]}       # would display “three”
```

Associative Arrays

An associated array is an array of elements referenced by a “key” value. Entries are added or updated using a key and value pair, and referenced (read) by using the “key” as the index into the array.

Unlike an indexed array an associative array must be explicitly defined as an associative array, failure to do so will see it treated as an indexed array with (as a key is normally non-numeric) only entry [0] in the array ever being updated; as seen in the below example.

Setting explicit type for variables is covered in the section on “setting specific types for variables”.

The below example shows what happens when an array is allowed to default and you try to use it as an associative array, as the key passed is non-numeric it always defaults to indexed entry 0.

```
[mark@phoenix]$ myarray["test1"]="line 1" # non-numeric so 0 is used
[mark@phoenix]$ typeset -p myarray        # show contents to confirm
declare -a myarray=([0]="line 1")
[mark@phoenix]$ myarray["test2"]="line 2" # non-numeric so 0 is used
[mark@phoenix]$ typeset -p myarray        # show contents to confirm
declare -a myarray=([0]="line 2")
[mark@phoenix]$
```

The correct ***and only*** way to use an associative array is to explicitly define it as an associative array before use, as seen in the below example.

```
[mark@phoenix]$ typeset -A myarray   # declare the array as associative
[mark@phoenix]$ myarray["test2"]="line 2" # non-numeric keys now allowed
[mark@phoenix]$ typeset -p myarray        # show contents to confirm
declare -A myarray=([test2]="line 2" )
[mark@phoenix]$ myarray["test1"]="line 1" # non-numeric keys ok(add new)
[mark@phoenix]$ typeset -p myarray        # show contents to confirm
declare -A myarray=([test2]="line 2" [test1]="line 1" )
[mark@phoenix]$ echo $myarray["test1"] # can NOT be accessed without {}
[test1]
[mark@phoenix]$ echo ${myarray["test1"]}  # can with {} syntax
line 1
[mark@phoenix]$
```

Reference Variables

For the purposes of this document it is just mentioned that reference variables exist.

A reference variable is a “name” that points to another existing variable, so a variable names “animal” could have reference variables such as “cat”, “dog”, “fish” pointing to it, so if “dog” was updated the actual effect is that “animal” gets updated.

You should never use reference variables in a script, as it has the effect of making the script unmaintainable and unreadable.

For that reason the only mention of reference variables in this document is, do not use them in scripts.

Setting Specific Types For Variables

The ‘bash’ shell uses ‘declare’ to explicitly set variable types, it also supports ‘typeset’ for backward compatibility.

Note: ksh and many other shells only typeset, so typeset should always be used if you wish you script to be portable. For that reason all examples here will use typeset.

This can be used in a variety of ways, the most common would be

- -i enforcing that a variable must be an integer
- -l changing all variable data to lowercase when data is supplied
- -u changing all variable data to uppercase when data is supplied

Examples

```
[mark@phoenix work]$ typeset -i aa    # only integers allowed in aa
[mark@phoenix work]$ aa=4
[mark@phoenix work]$ echo ${aa}
4
[mark@phoenix work]$ aa="A"           # non integers set aa to 0
[mark@phoenix work]$ echo ${aa}
0
[mark@phoenix work]$ aa=5
[mark@phoenix work]$ echo ${aa}
5
[mark@phoenix work]$ typeset -l bb    # bb to set all data input to
lowercase
[mark@phoenix work]$ bb="ABCD"
[mark@phoenix work]$ echo ${bb}
abcd
[mark@phoenix work]$ typeset -u bb    # bb to set all input to uppercase
[mark@phoenix work]$ echo ${bb}       # does not affect existing data
abcd
[mark@phoenix work]$ bb="abcd"        # but affects all new data entered
[mark@phoenix work]$ echo ${bb}
ABCD
[mark@phoenix work]$
```

A few other options you may see used in scripts are

- -r making a variable read only
- -a enforcing that a variable is an indexed array
- -A enforcing that a variable is an associative array

For other further options use ‘man bash’ and search on typeset.

You should also note that for variables “typeset -p variablename” can be used to describe a variable and its contents; for variables only (it will not work against functions for which the equivalent would be -f (or -f -F if you only want the name and not the entire function contents)). Examples of using both are scattered throughout the document in various examples.

Condition Tests

Condition tests are needed for tests in operations such as IF and WHILE.

The main condition tests you will use in a bash script are text matching and numeric value checks, and ***it is important to note that the comparison operators between the two types are different***.

- For string compares you would use “==” or “!=” for equal or not-equal.
- For numeric tests you would use -eq, -ne, -lt, -gt.
- If you tried to use == for a numeric comparison or -eq for a string comparison it would result in an error
- string compares must be within quotes
- numeric comparisons must not be within quotes

Condition tests can also include ‘ -a ‘ and ‘ -o ‘ between tests for AND and OR operations respectively.

I would also recommend for any string comparison tests to use an extra character in the tests to avoid errors if there is no data in a variable, for example ‘ [“${aa}X” != “sometextX”]’ rather than ‘[“${aa}” != “sometext”]’ as it variable aa is empty an error will be thrown. In scripts it may often occur that a text string can be empty if its value is to be obtained from commands being processed and this handles that condition. *In all the examples in this document and in real life I use the dot (.) character rather than X simply because for me it makes scripts more readable*.

You should not however perform similar methodology for numeric variables by prefixing a 0 before numeric variable tests, simply because a numeric variable should always have a value and if it does not something has gone seriously wrong in your script logic and you do want to have an error reported. If it is important that a numeric variable never be emoty (and the variable name is not used elsewhere for anything non-numeric) then define it as an integer as discussed in an earlier section on “setting specific types for variables” to ensure it will always contain a value, even if that value may be 0.

Examples

```
[ “${aa}.” != “yes.” -a “${aa}.” != “no.” ];     # if variable aa  not equal yes and not equal no
[ “${aa}.” == “yes.” -o “${aa}.” == “go.” ];     # if variable aa equal yes or equal go
[ “${aa}.” == “debug.” ];                        # if variable aa equal debug
[ ${aa} -lt 10 -o ${aa} -gt 20 ];                # if variable aa < 10 or > 20
```

The [condition] syntax parameter is not the only syntax available. The purpose of the [ ] or [[ ]] constructs in condition tests is to return 0 or 1 which may be provided by other tests also. Refer to the example in the “advanced while loop” section to see how this can be used.

IF, ELSE and ELIF

The syntax is fairly simple in bash, and is basically

```
if [ condition ];
then
   … some code ...
elif [ condition ];
then
   … some code ...
else
   … some code ...
fi
```

A critically important thing to note is if there is no executable code where '… some code ...' needs to be your script will error; not always with a meaningful error message. Do not just place comments in there as place-holders while developing a script, and 'echo "TODO"' would be OK but there must be executable code.

The main condition tests you will use in a bash script are text matching and numeric value checks.

The recommended practise in text condition tests is to include an extra character in the tests, this prevents errors where a variable may be empty. For example the below test will expand to a test of if ["X" == "YESX"], if the extra X was not used it would expand to if [== YES] which would cause a syntax error; resulting in the script simply falling through to following code with unexpected results.

```
userinput=””
if [ “${userinput}X” == “YESX” ];
then
   echo “User input is yes”
elif [ “${userinput}X” == “NOX” ];
   echo “User input is no”
else
   echo “User input is an unexpected value”
fi
```

Obviously your script will never have errors that result in an uninitialised variable (grin), but is is strongly recommended you use the extra character method coded in a way that the default if there is an empty variable is the preferred action path.

WHILE Loops

While loops are also quite simple in bash. The syntax is basically

```
while [ condition ];
do
...some code…
done
```

For example

```
aa=0
while [ ${aa} -lt 10 ];
do
   aa=$(( ${aa} + 1))
   echo ${aa}
done
```

Or, if the variable has been declared as an integer you could use

```
typeset -i aa=0 # must be declared as an integer (-i) for += syntax to work
while [ ${aa} -lt 10 ];
do
     aa+=1      # syntax can be used as aa declared as integer
     echo ${aa}
done
unset aa
```

While loops are always required if the condition you are checking against is a text match, for example

```
ready_to_exit=”NO”
while [ “${ready_to_exit}.” == “NO.” ];
do
    ...some code…
    ...some code that sets ready_to_exit to a different value…
done
```

Advanced WHILE Loops

The [condition] parameter is not the only syntax available. The purpose of the [ ] or [[ ]] constructs in condition tests is to return 0 or 1 which may be provided by other tests also.

Something commonly seen is use for ‘while’ loops is variations of the below example.
The while loop will in this example will continue as long as the pipe provides the ‘read’ command with input on STDIN; the ‘read’ command places the line read into the variable after the command, in the above case filename.

```
ls somedirectory | while read filename
do
    ...code to do something with filename...
done
```

An additional mention of the ‘read’ command is made in the “prompting for user input” section.

The while loop will take any list of strings when using the method above.

It is most often used in file operations, for example renaming files.

```
ls somedirectory | while read filename
do
   mv ${filename} ${filename}_backup
done
```

It is important to note that file names, or any variable that may contain any sort of special characters require correct handling; for example file names with spaces in the example above will break it.
In your scripts you need to be aware of when to use quoting.

FOR Loops

FOR Loops With a Counter

For loops of this type can only be done with arithmetic evaluations. If you want a loop based on text evaluation look at the while loop syntax.

This type of for loop requires the syntax below. ***It is important that you note the two bracket syntax for the for control statement***, it must use ((rules)), a single bracket syntax is an error.

```
for (( start-condition; end-condition; condition-modifier ));
do
   ...some code...
done
```

Example

```
for (( counter=0; counter < 10; counter=$((${counter} + 1)) ));
do
   echo “${counter}”
done
```

This is rather simple and the syntax is common across multiple languages so you must already be familiar with it, so we won’t waste any more time on explaining this one.

FOR Loops with a word list

This invocation of the for loop will run the code within the for loop for each item in the “in” list provided to the for command.

The item in the word list being processed is placed into the variable named in the statement, and can be used in the loop using that variable, in my examples here “myvar”.

The syntax for using the for command with a word list is simply (with a training semicolon (;) after the last list item)

This example will simply list word1, word2, word3, word4 and word5 on separate lines

```
for myvar in word1 word2 word2 word4 word5;
do
   echo “${myvar}”
done
```

CASE Statements

Case statements are used frequently in shell scripts, for starters they are the most common way of obtaining parameters passed to a script which is shown in the section on “obtaining parameters passed to your script” later on in this document.

A case statement comprises the value to be tested, and a list of possible values and the actions to take on a value match, and a default section, and the keyword ‘**esac**’ to end the ‘**case**’.

It is important to note that each section/test is terminated by a double semicolon (;;); that is required.

The basic syntax of a case statement is

```
case “${var}” in
   “val1”) ...code to run if var contained value1…
           ;;
   “val2”|”val3”) ...code to run if var contained val2 OR val3
           ;;
   “val4”) ...code to run if var contained val4
           ;;
   *)      ...some code if no values matched a defined entry…
           ;;
esac
```

For a good example on using the case statement see the section on “obtaining parameters passed to your script” later on in this document.

Obtaining any parameters passed to your script

A script is normally run as “scriptname parm1 parm2 parm3…”.
The entire command is treated as parameters so it is important to note that $0 is the name of the script being run.
So the actual parameters are passed to the script as variables $1, $2, $3, … and can be tested for by your script.

It is also important to note that $# will contain the number of parameters passed to the script (***it is modified by ‘shift’***).

The reserved keyword ‘shift’ is used to move parameters left, dropping the leftmost. So if your script was passed parm1 parm2 parm3 after the ‘shift’ command the parameters would be parm2 parm3. The ‘shift’ keyword as a result of dropping the leftmost parameter also decrements the $# value permitting it to be used as a loop control, and as the parameters shift what was $2 becomes $1 ($3 becomes $2 , … etc).

There are methods of ‘saving the stack’ and reverting it back after moving things about, however as 99% of scripts only use the parameters at script initialisation, *and you should avoid in scripts obscure functions that would make it difficult for people that follow you from maintaining the script*, they will not be covered here. If you wish to maintain the stack as $# provides the number of parameters you can always use a for loop rather than a while loop to obtain them and avoid the ‘shift’ command.

It is important to mention the use of the ’shift’ command however, as you may wish to use it in functions you write.

This is an example of a common way to obtain parameters passed to a script. All parameters passed are expected to be in the format “keyword=value” (or in my case always “--keyword=value” as it is more easily identifiable to a script user that it is a parameter).

```
MAX_NUMBER=0          # default is 0
YES_OR_NO="no"        # default is no
ANY_VALUE=""          # default is nothing
PARM_ERRORS="no"
while [[ $# -gt 0 ]];
do
   parm=$1
   key=`echo "${parm}" | awk -F\= {'print $1'}`
   value=`echo "${parm}" | awk -F\= {'print $2'}`
   case "${key}" in
      "--maxnum")  testvar=`echo "${value}" | sed 's/[0-9]//g'`
                   # the above sed command strips out all numerics
                   # so if anything is left the value was not numeric
                   if [ "${testvar}." != "." ];
                   then
                      echo "*error* the --maxnum value provided is not numeric"
                      PARM_ERRORS="yes"
                   fi
                   MAX_NUMBER="${value}"
                   shift
                   ;;
      “--anotherparm”)  ANY_VALUE="${value}"
                 shift
                 ;;
      "--yesno") if [ "${value}." != "yes." -a "${value}." != "no." ];
                 then
                    echo "*error* the --yesno value provided is not yes or no"
                    PARM_ERRORS="yes"
                 fi
                 YES_OR_NO="${value}"
                 shift
                 ;;
      *)         echo "*error* the parameter ${key} is not a valid parameter"
                 PARM_ERRORS="yes"
                 shift
                 ;;
   esac
done
… more sanity tests on parms, if any bad update PARM_ERRORS
if [ "${PARM_ERRORS}." != "no." ];
then
   echo "Script aborted due to the above errors"
   echo "Please read the documentation."
   exit 1
fi
```

Using your own functions in a script

Creating user functions

Functions can be explicitly defined using the “declare” or “typeset” commands, but the simplest way is simply to define one, bash is smart enough to know that a variable defined as
“varname() { }”
is a function, of course you should actually put function code within the { } at the time you define it.

User script functions can be passed parameters which can be retrieved by the function in the same way as user parameters passed to a bash script file, using the $1, $2,… syntax. The ‘shift’ operation can also be used within user functions.

For example

```
#!/bin/bash
# define a function, call it a few times
myfunction() {
parmvalue=”$1”
echo “”Got parm: ${parmvalue}”
}

myfunction “line 1”
myfunction “line2”
exit 0
```

The return function

An implicit return is assumed when the end of the function is reached so it does not need to be coded.

The “return” command can be used anywhere within the man body of the function to return from the function immediately, with the following warnings; the “return” command can also be used within for loops and while loops to immediately exit those loops so if your function uses those and you use the “return” command from within those the function will not exit as you expect so you will need handling outside those loops in your function to determine if you want to exit the function as well as the loop.

Testing if a function exists

It is possible to test if a function exists using the bash “typeset” function. There are some considerations to be aware of as shown in the examples below.

- “typeset -f <name>” will return the function contents, “typeset -f -F <name>” will return only the function name
- if the function exists the return code will be 0, if it does not exist the return code will be 1
- if using “typeset -f ...” against a variable rather than a function as it is not a function the return code will be 1
- if you wish to test for a variable rather than a function you can use the -p option, but note that a function is not considered a variable so using -p will return 1 if used against a function

Example run through

```
# create a simple function
[mark@phoenix work]$ myxx() { return; }

# test if the function exists, if yes result code is 0
# be default the entire function is displayed
[mark@phoenix work]$ typeset -f myxx;echo $?
myxx ()
{
    return
}
0

# test if the function exists, if yes result code is 0
# using -F only the name of the function is returned
# and not the entire contents
[mark@phoenix work]$ typeset -f -F myxx;echo $?
myxx
0

# if the function does not exist rge result code is 1
[mark@phoenix work]$ typeset -f -F nofunc;echo $?
1

# if a variable name is passed the result code is 1
# as a normal variable is not a function
[mark@phoenix work]$ aa="test"
[mark@phoenix work]$ typeset -f aa;echo $?
1

# note also that typeset -p can display the definition of a variable
[mark@phoenix work]$ typeset -p aa;echo $?
declare -- aa="test"
0

# but typeset -p cannot be used of a function, as a function
# is not a variable so will not be found.
[mark@phoenix work]$ typeset -p myxx;echo $?
```

```
bash: typeset: myxx: not found
1
```

Another important consideration in testing for a function existing within a script is in the format of the actual if statement itself.

The below will work correctly

```
if [ `typeset -f -F myxx > /dev/null; echo $?` == 0 ];
then
   echo "Var myxx IS a function"
fi
```

The below will not work correctly, running the below from a file with “bash -x” shows the evaluation is correctly “if [‘ 0 ‘];” but it is not treated as a 0 and will always report not a function if used in a if statement this way, so always use the above method.

```
if [ `typeset -f -F myxx > /dev/null; echo $?` ];
then
   echo “Var myxx is not a function”
else
   echo "Var myxx IS a function"
fi
```

The reason you may wish to test for the existence of a function is to determine if a required function library file needed by a script has already been loaded and if the function test fails source in the library file; although a well written script will never need such a test.

Never put such tests in library files themselves, an endless loop is the most likely result of doing that.

Handling system utility (and script function) output

Piping into a while loop

If you have read this document from the beginning you will already have seen how this can be achieved.

If you have forgotten refer to the “advanced while loops” section again.

Placing command output into a variable

The output of commands can be placed into a variable using a pair of the back-quote (sometimes referred to as the back-tick character (`) found on US keyboards above the left tab key (on the key with the shifted ~ character).

You will the use of this redirection in most scripts as it will place all multi-line output of a command into the variable; which is generally more useful that passing data through a pipe to something like ‘read’ which only accepts one line at a time.

The syntax is also extremely simple.
variablename=`command and parms to execute`

For example the below would place the contents of a “ls -la” command into variable filelisting; it cannot get much simpler.

```
filelisting=`ls -la`
```

Formatting is preserved in the variable, if it was a multi-line response then operations such as ‘echo’ to the screen or redirected to a file will be the unchanged multi-line response. Likewise an echo of the variable to a pipe would be the same as if the ls -la command was being sent to the pipe.

This is an efficient way of obtaining both one line responses and the output of a command you may want to process more that once (it is more efficient to use the buffered output multiple times than keep rerunning the command).

It is also important to note that this syntax can also be used to redirect and capture the output of functions as well as unix commands.

Note that data obtained this way is not an array, it is a single simple variable with text containing line breaks; as such the variable can be “echo”ed into a while loop and processed exactly as if the command itself was run.

If you want the data in an array rather than a simple text variable refer to the section on “indexed arrays” which mentions how to load multi-line data into an indexed array.

More advanced, and in most cases pointless

Asynchronous processing is not yet fully implemented

The “bash” shell partially supports asynchronous processing, where a ‘subshell’ can be started in the background to run commands (or even functional server tasks) with the subshell standard input and standard output available to the initiating bash shell.

While file descriptors are made available in the initating shell to provide stdin and stdout file descriptors to the subshell so it can be fully utilised as a ‘server’ task by the initiating shell.

However a current limitation in the bash shell is that it can start at most one coproc subshell (unless you wish to compile it from source using experimental features), making bash unsuitable for an application that needs co-processing functionality.

If you are interested in this feature of bash as a starting point you should “man bash” and search on the word “coproc” for a full description. Also you may wish to look at the ‘ksh’ shell if you can live without any of the bash builtin variable functions, many of which ksh does not support.

User interaction with your scripts

Why not to have user interaction

Scripts are generally for complex batch tasks, many intended to run from under cron or another scheduler. No script that is intended to run as batch should ever prompt for user input as if it runs under a scheduler it may just hang forever waiting for that input.

Unfortunately some system utilities that may be run by scripts do prompt for input, for example tar which wants to confirm if it is ok to overwrite an existing tar file. If that happens under cron the command can hang forever (I modified the job scheduler I wrote for my needs to handle such conditions but most *nix schedulers cannot).

If a script is well written then all operations the script can perform should be able to be controlled by parameters passed to the script to control its operation.

If however there must be places in the script that might require user intervention the script must be able to accept parameters passed with default answers to allow the script to run correctly in a batch environment.

If however your script is designed to run with user interaction the following sections can assist with that.

The ‘read’ statement

The simple read statement

You have arealy seen an example of the use of ‘read’ in a batch mode in the “advanced while loops” section, where the ‘read’ command was used to process input passed to it via its STDIN.

The ‘read’ statement also allows a script to interactively prompt for user input, and place the response into a variable.

As the ‘read’ statement when used this way is just a one line prompt for input you should always display additional text to the user prior to the prompt to explain why they are being prompted.

The syntax of the ‘read’ statement is best explained by this simple example

```
read -p "Do you wish to continue (y/n)?" testvar
```

The user is prompted with the text provided by the -p option, and the user response is placed into the variable named, in the example above the response is placed into the variable testvar.

The user may enter any data they wish to, which means you must sanitise the user data before risking variable expansion in case they may have accidentally input escape characters that would cause a command to execute when the variable is expanded. And yes I do mean accidentally, as the user running the script already has authority to do whatever they wish, running a script gives them no more authority (unless you stupidly setuid the script).

Also unless you explicitly limit the amount of data that can be read (as seen in the below example) you have to be wary of buffer overflows; *although some implementations of read accept a maximum of 128 characters*.

But any user input entered via ‘read’ needs to be checked and sanitised. In the simple case above where only y/n is permitted that can be done simply by ensuring the length of the variable is 1 (which can be enforced by the read command as seen in the below example) then testing for y or n, but in cases where the user input can be more complex additional checking is required.

Basically you cannot trust uncontrolled user input to a script, so ‘read’ it should be avoided (disclaimer: the author of this document does use it, but only for Y/N answers where absolutely necessary).

To see all the options available to ‘read’, as always refer to ‘man bash’.

The ‘read’ command is more often used for non-interactive use; an example of which is seen on the “advanced while loops” section.

Limiting the length of data read

The ‘read; command does have many options, for example the y/n example shown above could be replaced with the below to limit user input to one character with the “-n” option; “-n 1” is accept only one character

```
read -n 1 -p "Do you wish to continue (y/n)?" testvar
```

Setting a timeout for user input

Or perhaps replaced with the below to also timeout after 120 seconds using the “-t” flag.
This is a little more complicated as it provides exit status codes to check; exit status 0 is data provided, >0 and <=128 no data entered, > 128 timeout ocurred)

```
read -n 1 -t 120 -p "Do you wish to continue (y/n)?" testvar
result=$?
if [ ${result} -gt 128 ];
then
   echo “Timeout reached, using defaults”
   testvar=”some default value”
elif [ ${result} -gt 0 ];
then
   echo “No data entered, using default”
   testvar=”some default value”
# else user input was provided so testvar is set
fi
echo “Got value ${testvar}”
```

The select statement

The select statement is the preferred way of obtaining known/expected user input as the user is only permitted to provide hard coded responses and eliminates some of the risks associated with users being able to type in whatever they want.

The 'select' command displays a numbered list containing the words provided.

- If a valid number is entered the variable name provided is set to the word selected
- if EOF is used the select exists; *normal exit would be if the 'break' command is used the select exits*
- user input is also stored in variable REPLY
- a custom prompt can be set in PS3
- if the user enters nothing the list is just re-displayed
- **a bad thing** is that if the user enters a number not in the list then '...some code...' is still run but with a null value in the select variable; it is up to the coder to handle that; generally with a 'case' statement handling the option the user selected

The basic syntax of the select command is as below

```
select name in word1 word2 word3 word4;
do
    ...some code...
done
```

Example

```
select name in “word” “list” exit;
do
    case “${name}” in
        ”word”|”list”)          # if word or list were selected
             echo “selected ${name}”
            echo “user input was ${REPLY}”
            ;;
         “exit”)
            break
           ;;
        *)  # ignore invalid responses
    esac
done
```

If you must prompt the user for input using 'select' is probably the safest way if explicit responses are required as the user can only use the data values you provide

BASH built in functions you will use often

Simple Arithmetic

The ‘bash’ shell provides a inbuilt arithmetic funtion obtained by simply using the syntax “$((your calculation))”, and yes two brackets on either side is required. Variables may also be used within the calculation.

The arithmetic function follows the normal rules of using brackets to control computation, as seen in the examples below.

Examples

```
[mark@phoenix]$ mynum=100
[mark@phoenix]$ echo $(( ${mynum} * 10 ))
1000
[mark@phoenix]$ echo $(( 100 - (${mynum} / 50) ))
98
[mark@phoenix]$ echo $(( 100 - (${mynum} - 50) ))
50
[mark@phoenix]$ mynum2=50
[mark@phoenix]$ echo $(( ${mynum} - ${mynum2} ))
50
[mark@phoenix]$
```

Obtaining the variable length

The length of a variable is obtained with the ${#variablename} syntax.

For example “aa=’ABCD’;echo ${#aa}’” would display 4.

That is so simple we won’t spend any more time on that here.

Extracting a Substring from within a string

The bash shell provides many variable operations, one of the most useful is the variable substring function. As this is a built-in variable operation within bash it is the most powerful differentiator between bash and other shells; other shells need to use external utilities to perform this function.

The syntax is simply **${variablename:startoffset:lengthtocopy}**; the only thing to note is that the start offset begins at zero.

Examples, note that the last example uses the length function mentioned above to show that if the length is longer than the variable length remaining it is happily truncated.

```
[mark@phoenix work]$ xx="ABCDEF"
[mark@phoenix work]$ yy=${xx:0:2}
[mark@phoenix work]$ echo ${yy}
AB
[mark@phoenix work]$ yy=${xx:3:2}
[mark@phoenix work]$ echo ${yy}
DE
[mark@phoenix work]$ yy=${xx:4:${#xx}}
[mark@phoenix work]$ echo ${yy}
EF
```

As noted in the section title this function is only available in the BASH shell, not in most other shells. Therefore use of this function will make your script non-portable to other environments.
I do not consider this an issue

Extracting data to the left or right of a search substring key

There will be occasions where you want to not just search for the existence of a substring, but wish to extract the data only on one side of the substring. This can be achieved in bash without the need for complex awk or egrep statements.

Testing if the substring exists in the string to be parsed

If the substring you are searching on is not in the data string you are searching on the result of the below two operations will generally be the entire data string. It is therefore wise to always test if a substring actually exists in the data string to be parsed.

As bash does not have an ‘index’ type function to return the position of a substring to use as a test the best way of achieving a valid test is as in the example below.

```
datastr=”This is a string that contains this is the key in the data”
parmkey=”this is the key”
if [[ $datastr == *"${parmkey}"* ]];
then
   echo “The data string contains the search key ${parmkey}”
else
   echo “the data string does not contain the search string ${parmkey}”
fi
```

Obtaining all the text after a keyword

The main reason you would want to locate a substring within a string is to obtain the text before or after the substring.

This is how to obtain the string data after a substring you are searching for.

```
datastr=”This is a string data that contains this is the key in the data”
parmkey=”this is the key”
# rest will contain all data after the matched substring
rest=${datastr#*$parmkey}
```

rest will contain: in the data

Obtaining all the text before a keyword

The main reason you would want to locate a substring within a string is to obtain the text before or after the substring.

This is how to obtain the string data before the substring you are searching for.

```
datastr=”This is a string data that contains this is the key in the data”
parmkey=”this is the key”
# rest will contain all data before the matched substring
rest=${datastr%$parmkey*}
```

rest will contain: `This is a string data that contains`

Obtaining the index of a substring within a string

As mentioned earlier bash does not provide an index function to locate the offset of a substring within a string. As it does provide as mentioned above methods of extracting the string data before and after a substring, which is normally why you would want to locate a substring, it is not really required.

If you require that function it is easy to implement using the inbuilt functions described above.

1. test the substring exists in the string
2. into a temporary variable place the data before the substring
3. use the length of the temporary variable to obtain the offset it the substring was in the string

That is an exercise you can to to test if you have read everything above and read the section on user defined functions.

Forcing all variable contents to upper or lower case

This was covered in the section on **setting specific types for variables**, where a variable can be declared as containing upper or lower case data. If a variable is declared in this way all data values used to set/modify the variable will be converted to the correct case by bash when the variable is modified.

If you skipped over the section on setting specific variable types you may want to refer back to it.

If you have not explicitly defined variables as upper or lower case, or wish portability, you will see most scripts use the external “tr” utility to convert between upper and lower case.

Common utilities used in shell scripts

AWK – selecting specific fields from data

```
awk '{print $NF}'   prints the last field/word
```

SED – string character editing

TR – character translation
mention uppercase/lowecase special case

TAIL/HEAD

MORE STUFF