Bash Shell Scripting - 10 Seconds Guide

October 23, 2005 Posted by Ravi

This **Bash shell scripting** guide is not a detailed study but a quick reference to the BASH syntax. So lets begin...

> If you like this 10 seconds guide, don't forget to read -
> - Bash Shell Shortcuts
> - Special Shell Variables

Common environment variables

`PATH` - Sets the search path for any executable command. Similar to the PATH variable in MSDOS.

`HOME` - Home directory of the user.

`MAIL` - Contains the path to the location where mail addressed to the user is stored.

`IFS` - Contains a string of characters which are used as word seperators in the command line. The string normally consists of the space, tab and the newline characters. To see them you will have to do an octal dump as follows:

```
$ echo $IFS | od -bc
```

`PS1` and `PS2` - Primary and secondary prompts in bash. PS1 is set to `$` by default and PS2 is set to `>` . To see the secondary prompt, just run the command :

```
$ ls |
```

... and press enter.

`USER` - User login name.

`TERM` - indicates the terminal type being used. This should be set correctly for editors like Vim to work correctly.

`SHELL` - Determines the type of shell that the user sees on logging in.

> To see what are the values held by the above environment variables, just do an `echo` of the name of the variable preceded with a `$`.

For example, if I do the following:

```
$ echo $USER
```

```
ravi
```

... I get the value (My login name) which is stored in the environment variable `USER`.

Some bash shell scripting rules

- The first line in your script must be `#!/bin/bash`
  ... that is a `#` (Hash) followed by a `!` (bang) followed by the path of the shell. This line lets the environment know the file is a shell script and the location of the shell.
- Before executing your script, you should make the script executable. You do it by using the following command:

```
$ chmod ugo+x your_shell_script.sh
```

- The name of your shell script must end with a `.sh` . This lets the user know that the file is a shell script. This is not compulsary but is the norm.

Conditional statements

'if' Statement

The 'if' statement evaluates a condition which accompanies its command line.
syntax:

```
if condition_is_true
then
   //execute commands
else
  //execute commands
fi
```

'if' condition also permits multi-way branching. That is you can evaluate more conditions if the previous condition fails.

```
if condition_is_true
then
   //execute commands
elif another_condition_is_true
then
   //execute commands
else
   //execute commands
fi
```

Example :

```
if grep "aboutlinux" thisfile.html
then
   echo "Found the word in the file"
else
   echo "Sorry no luck!"
fi
```

if's companion - test

`test` is an internal feature of the shell. 'test' evaluates the condition placed on its right, and returns either a true or false exit status. For this purpose, 'test' uses certain operators to evaluate the condition. They are as follows:

Relational Operators

- `-eq` - Equal to
- `-lt` - Less than
- `-gt` - Greater than

- `-ge` - Greater than or Equal to
- `-le` - Less than or Equal to

File related tests

- `-f file` - True if file exists and is a regular file.
- `-r file` - True if file exists and is readable.
- `-w file` - True if file exists and is writable.
- `-x file` - True if file exists and is executable.
- `-d file` - True if file exists and is a directory.
- `-s file` - True if file exists and has a size greater than zero.

String tests

- `-n str` - True if string str is not a null string.
- `-z str` - True if string str is a null string.
- `str1 == str2` - True if both strings are equal.
- `str` - True if string str is assigned a value and is not null.
- `str1 != str2` - True if both strings are unequal.
- `-s file` - True if file exists and has a size greater than zero.

Test also permits the checking of more than one expression in the same line.
- `-a` - Performs the AND function
- `-o` - Performs the OR function

A few Example snippets of using test

```
test $d -eq 25 && echo $d
```

... which means, if the value in the variable d is equal to 25, print the value. Otherwise don't print anything.

```
test $s -lt 50 && do_something
```

```
if [ $d -eq 25 ]
then
echo $d
fi
```

In the above example, I have used square brackets instead of the keyword test - which is another way of doing the same thing.

```
if [ $str1 == $str2 ]
then
    //do something
fi

if [ -n "$str1" -a -n "$str2" ]
then
    echo 'Both $str1 and $str2 are not null'
fi
```

... above, I have checked if both strings are not null then execute the echo command.

Things to remember while using test

1. If you are using square brackets `[ ]` instead of `test`, then care should be taken to insert a space after the `[` and before the `]`.
2. `test` is confined to integer values only. Decimal values are simply truncated.
3. Do not use wildcards for testing string equality - they are expanded by the shell to match the files in your directory rather than the string.

Case statement

Case statement is the second conditional offered by the shell.
Syntax:

```
case expression in
pattern1) //execute commands ;;
pattern2) //execute commands ;;
...
esac
```

The keywords here are in, case and esac. The ';;' is used as option terminators. The construct also uses ')' to delimit the pattern from the action.

Example:

```
...
echo "Enter your option : "
read i;

case $i in
1) ls -l ;;
2) ps -aux ;;
3) date ;;
4) who ;;
5) exit
esac
```

> The last `case` option need not have `;;` but you can provide them if you want.

Here is another example:

```
case `date |cut -d" " -f1` in
Mon) commands ;;
Tue) commands ;;
Wed) commands ;;
...
esac
```

Case can also match more than one pattern with each option.You can also use shell wild-cards for matching patterns.

```
...
echo "Do you wish to continue? (y/n)"
read ans

case $ans in
Y|y) ;;
[Yy][Ee][Ss]) ;;
N|n) exit ;;
[Nn][Oo]) exit ;;
*) echo "Invalid command"
esac
```

In the above case, if you enter YeS, YES,yEs and any of its combinations, it will be matched.

This brings us to the end of conditional statements.

Looping Statements

while loop

`while` loop syntax -

```
while condition_is_true
do
   //execute commands
done
```

Example:

```
while [ $num -gt 100 ]
do
sleep 5
done
```

```
while :
do
   //execute some commands
done
```

The above code implements a infinite loop. You could also write 'while true' instead of 'while :' .
Here I would like to introduce two keywords with respect to looping conditionals. They are *break* and *continue*.

`break` - This keyword causes control to break out of the loop.

`continue` - This keyword will suspend the execution of all statements following it and switches control to the top of the loop for the next iteration.

until loop

`until` complements `while` construct in the sense that the loop body here is executed repeatedly as long as the condition remains false.

Syntax:

```
until false
do
   //execute commands
done
```

Example:

```
...
until [ -r myfile ]
do
sleep 5
done
```

The above code is executed repeatedly until the file myfile can be read.

for loop

`for` loop syntax :

```
for variable in list
do
   //execute commands
done
```

Example:

```
...
for x in 1 2 3 4 5
```

```
do
   echo "The value of x is $x";
done
```

Here the list contains 5 numbers 1 to 5. Here is another example:

```
for var in $PATH $MAIL $HOME
do
   echo $var
done
```

Suppose you have a directory full of java files and you want to compile those. You can write a script like this:

```
...
for file in *.java
do
   javac $file
done
```

> You can use wildcard expressions in your scripts.
> Read Regular Expressions Tutorial to know more.

Special symbols used in BASH scripting

- `$*` - This denotes all the parameters passed to the script at the time of its execution. Which includes `$1`, `$2` and so on.
- `$0` - Name of the shell script being executed.
- `$#` - Number of arguments specified in the command line.
- `$?` - Exit status of the last command.

The above symbols are known as positional parameters. Let me explain the positional parameters with the aid of an example.

Suppose I have a shell script called my_script.sh . Now I execute this script in the command line as follows :

```
$ ./my_script.sh linux is a robust OS
```

... as you can see above, I have passed 5 parameters to the script. In this scenario, the values of the positional parameters are as follows:

- `$*` - will contain the values 'linux','is','a','robust','OS'.
- `$0` - will contain the value `my_script.sh` - the name of the script being executed.
- `$#` - contains the value 5 - the total number of parameters.
- `$$` - contains the process ID of the current shell. You can use this parameter while giving unique names to any temporary files that you create at the time of execution of the shell.
- `$1` - contains the value 'linux'
- `$2` - contains the value 'is'

... and so on.

The set and shift statements

`set` - Lets you associate values with these positional parameters .

For example, try this:

```
$ set `date`
$ echo $1
$ echo $*
$ echo $#
$ echo $2
```

`shift` - transfers the contents of a positional parameter to its immediate lower numbered one. This goes on as many times it is called.

Example :

```
$ set `date`
$ echo $1 $2 $3
$ shift
$ echo $1 $2 $3
$ shift
$ echo $1 $2 $3
```

To see the process Id of the current shell, try this:

```
$ echo $$
2667
```

Validate that it is the same value by executing the following command:

```
$ ps -f |grep bash
```

Make your BASH shell script interactive

read statement

Make your shell script interactive. `read` will let the user enter values while the script is being executed. When a program encounters the read statement, the program pauses at that point. Input entered through the keyboard id read into the variables following read, and the program execution continues.

An example -

```
#!/bin/sh
echo "Enter your name : "
read name
echo "Hello $name , Have a nice day."
```

Exit status of the last command

Every command returns a value after execution. This value is called the exit status or return value of the command. A command is said to be `true` if it executes successfully, and `false` if it fails. This can be checked in the script using the `$?` positional parameter.

Resources for more detailed study of the BASH command

Linux Shell Scripting Tutorial @ Cyberciti.biz
Introduction to BASH Programming @ Tldp.org
BASH FAQ @ Greg's Wiki
BASH Pitfalls @ Greg's Wiki
Unix shell scripting resource @ Shelldorado

I hope you enjoyed reading this **Bash shell scripting** 10 seconds guide.

Labels: bash shell , system administration

Newer Post Home Older Post

Back to Top