Shell scripting
Scripting and Computer Environment - Lecture 5

Saurabh Barjatiya

International Institute Of Information Technology, Hyderabad

28 July, 2011

Contents

Contents

Introduction

Shell scripting with bash is very powerful. Many programs / tools / utilities are coded in shell at begining of project and later on same code in written in C, C++ etc. for efficiency after project turns out to be useful.

Note: Previous knowledge of shell commands, regular expressions and experience in some programming language is assumed in this lecture.

When not to use shell scripting

One shoud avoid using shell scripting for:

- CPU intensive processing
- Programs that require complex math
- Security is very important
- Mission critical applications
- GUI programs
- Access to hardware like Joystick, Camera etc. is required
- Closed source applications :)

Creating scripts

- First line of shell script must indicate that they should be interpreted by shell. Full path of program to be used to interpret the script has to be specified followed by special sequence - #! (read as sha bang).
- If we want to execute script as program then we must give it execute permissions using 'chmod +x' command
- We can execute shell scripts by passing them as command line arguments to shell executables (/bin/bash). In this case they do not need to have execute permissions set.

Hello World script

Hello World script

```
#!/bin/sh

# This is simple hello world script.
# We can write comments on lines
# starting with hash (#) character

echo Hello World!
exit 0

# It is good habit to return 0 if
# script ends successfully.
```

Refer sample file: `01-hello_world.sh`

Uncommon interpreters

Any program that can take filename as input and does something with that file can be chosen as interpreter.

For example since we know ‘`cat`’, ‘`more`’, ‘`less`’, ‘`grep`’ etc. can accept filenames any of them can be used as interpreter.

Refer sample files: `02-more.sh`, `03-less.sh`, `04-create_rm.sh` and `05-grep.sh`.

Contents

Variables

- We do not need to declare variables in shell script. Variables automatically get created when they are assigned some value.
- Shell variables do not have any type like int, char, float etc.
- If we try to use shell variable without declaring then it is treated as null.
- We use dollar('$') to access value of shell variable. We do not use $ while setting value.
- Shell variables are case sensitive.

Refer sample file: `06-shell_variables.sh`

Quotes

- Variables get expanded in double quotes (“ ”) and do not get expanded in single quotes (‘ ’).
- When using double quotes (“ ”) dollar (‘$’) can be escaped using ‘\$’.

Refer sample file: `07-quotes.sh`

Arrays

- Shell supports arrays which can take integral indexes.
- Even non-array variables can be treated as arrays and their values can be accessed at index 0.
- The entire '<variable_name>[<index>]' pattern needs to be enclosed in curly braces '({})' when using arrays.

Refer sameple file: `08-arrays.sh`

Taking Input

- We can take input from user using 'read' keyword.
- read is followed by variable name. While using read, we do not prepend variable name with dollar ('$') sign.
- read supports options like timeout, read from particular file descriptor etc.

Refer sample file: 09-read.sh

Special characters

Shell or shell scripts have many different special characters and sequences. Some of the important characters that would get discussed in this lecture are:

Hash (#)	Semi-colon (;)	Dot(.)
Double-quote(‘ ‘)	Single-quote(’)	Comma(,)
Back-slash(\)	Back-tick (`)	Colon(:)
Asterisk(*)	Question mark(?)	Hypen(-)
Tilde(˜)		

Special characters - Hash (#)

- # can be used to start comments in a shell script.
- # can be used anywhere on the line. It is not necessary for it to be first character on first column of line. Whatever follows # on any line will get treated as comment.
- Space before # is important so that # does not gets some other meaning and signifies that rest of the line is comment
- To special # literally it can be escaped with '\'
- # works even on command line. But there are very limited reasons to specify comments on command line.

Refer sample file: `10-special_characters-hash.sh`

Special characters - Semicolon (;)

- Semicolon (;) can be used to specify more than one commands in the same line.
- It gets used very often in separating test condition from `then` keyword in `if` statements. Similarly it is used to separate test condition from `do` keyword in while loops.
- Double semicolon (;;) are used to terminate case option when using case.

Refer sample file: `11-special_characters-semicolon.sh`

Special characters - Dot(.) and Quotes (' ' and ")

- Quotes as discussed before can help in preserving meaning of special characters. Double quotes treat $ as special character and single quotes are even more strict and treat $ also as literal $.
- Single dot (.) can be used to denote current directory and double dots(..) can be used to denote parent directory.
- Dot followed by command (specially script) means source script. In this case the script will be run in same environment and shell.

Refer sample files: `12-special_characters-dot.sh` and `12-special_characters-dot-helper.sh`

Special characters - Comma(,)

- Comma when used in arithmetic expressions evaluates the value of entire comma expression as value of last expression. All expressions separated by comma get evaluated but only value of last expression is returned as return value / value of entire expression.
- Comma can be used to combine strings on shell. For example 'a{b,c}d' evaluates to 'abd acd'.

Refer sample file: `13-special_characters-comma.sh`

Special characters - Backslash(\)

- Backslash(\) is used to escape special characters like single quote ('), double quotes (") and even itself.
- Backslash can also be used as line continuation character. Whenever a line ends with backslash it indicates that same command is continued over next line too.

Refer sample file: `14-special_characters-backslash.sh`

Special characters - Back-tick (`)

- Back-tick can be used to give output of one command as comman line arguments to other command when used on shell.
- The same can also be used to capture output of a shell command in variable for further operations.

Refer sample file: `15-special_characters-backtick.sh`

Special characters - Colon (:)

- It is shell synonym for true
- It can be used as null statements in if
- It can be used to create anonymous here documents
- It can be used to truncate / create files.

Refer sample file: `16-special_characters-colon.sh`

Special character - Asterisk(*)

- Asterisk(*) can used as wild card to match anything include null
- Asterisk(*) can be used as multiplication operator
- Double asterisk(**) can be used for exponentiation

Refer sample file: `17-special_characters-asterisk.sh`

Special character - Question mark(?)

- Can be used as ternay operator (test?true:false) in mathematical expressions.
- Can be used as wild card to match any single character
- Can be used to check if some variable is set or not

Refer sample file: `18-special_characters-question_mark.sh`

Special character - Hypen(-)

- Hypen can be used for supplying default values to parameters
- Hypen are used to pass various types of command line arguments.
- Hypen can be used as filename for stdin/stdout
- Hypen can be used to go to $OLDPWD when supplied to cd as argument
- Hypen can also be used for arithmetic subtraction

Refer sample file: `19-special_characters-hypen.sh`

Special character - Tilde (~)

- Tilde can be used to find home folders of various users
- It can be combine with + or - to find $PWD or $OLDPWD respectively.

Refer sample file: `20-special_characters-tilde.sh`

Parameter Substitution - ${}

- Parameter substitution can be used to get values of variables
- It can be used to check if variable is set / not set and change / use values accordingly.
- It can be used to remove longest/shortet pattern match from beggining / end of value
- It can be used for search/replacement.
- It can also be used for extracting substrings from variables

Refer sample files: `21-parameter_substitution.sh` and `22-pattern_matching_examples.sh`

Special Variables

Shell has many special variables like:

- FUNCNAME, CDPATH, SECONDS etc. which store important values that can help in acheiving complex outputs / effects using shell scripts.
- $1, $2, $3, etc. can be used to get parameters passed to script or function. All parameters can be accessed at same time using $@ or $*
- $_, $?, $$, etc. help in getting part of arguments, exit status PID etc. which help in ensuring whether previous command got completed successfully, creating temporary files, etc.

Refer to sample file: `23-special_variables.sh`

if statement

- If statements can be used compare strings
- If statements can also work on numbers
- If statements can be used check permissions or file type
- Shell supports if, elif, else ladders
- If supports use of && or || for short-circuit AND or OR operations

Refer sample file: `24-if_statement.sh`

loops

- There are three types of loops in shell for, while and until
- for loop is like for-each loop used in other languages
- while and until are very similar and we can achieve one with help of other by using not (!)
- break and continue work in loops

Refer sample file: `25-loops.sh`

case statement

- Case statements (also referred to as switch cases in other languages) are very powerful constructs in shell scripting as unlike most languages in which cases take constants and sometimes just integers, shell scripts allows case values to be patterns.
- We can also specify more than one pattern and if any of the multiple patterns specified match the corresponding commands are executed.

Refer sample file: `26-case_statement.sh`

functions

- Shell functions are like normal functions in any other language. They support arguments and can return values.
- Functions work same way as other shell commands.
- The values returned cannot be directly assinged to variables. We have to use $? construct to capture returned value.
- We can store output (not return value) of function in variables.

Refer sample file: `27-function_basics.sh`

Miscellaneous

- Input / Output of most code blocks can be redirected
- Sub-shells can be used to group commands together in one single sub-shell. These sub-shells can even run in background
- <() can be used to redirect processes, which kind of means use output of process as filename
- Here documents allow convinient way of supplying input without creating temporary files.
- All variables in functions are global unless specified otherwise

Refer sample file: `28-miscellaneous.sh`