CS2043 - Unix Tools & Scripting
Lecture 9
Shell Scripting

Spring 2015 [1]

Instructor: Nicolas Savva

February 9, 2015

[1] based on slides by Hussam Abu-Libdeh, Bruno Abrahao and David Slater over the years

Announcements

- Coursework adjustments
  (now 4 assignments plus a final project)
- A3 is out (due Friday 02 / 20)

Scripting

This week we will discuss bash scripting. Before we begin, we will discuss a few preliminaries.
(and quickly review a few things for the sake of completeness)

Today's agenda:

- Shell variables
- Shell expansion
- Quotes in bash
- Running commands sequentially & exit codes
- Your "first" script
- Passing arguments to scripts
- If conditionals

Variables

- Bash scripting is very powerful!
- To get anything done we need variables.
- To read the values in variables, precede their names by a dollar sign ($).
- The contents of any variable can be listed using the `echo` command
- Two types of variables: Local and Environment.

Example:

```
echo $SHELL
/bin/bash
```

Local Variables

Local variables exist only in the current shell:

Example:

```
nsavva@maxwell:~$ x=3
nsavva@maxwell:~$ echo $x
3
```

Note: There cannot be a space after the x nor before the 3!

Environment Variables

- Environment Variables are used by the system to define aspects of operation.
- The Shell passes environment variables to its child processes
- Examples:
  - $SHELL - which shell will be used by default
  - $PATH - a list of directories to search for binaries
  - $HOSTNAME - the hostname of the machine
  - $HOME - current user's home directory
- To get a list of all current environment variables type `env`

New Environment Variable:

To set a new environment variable use `export`

```
nsavva@maxwell:~$ export X=3
nsavva@maxwell:~$ echo $X
3
```

Note: NO Spaces around the = sign.

A Word About the Difference

The main difference between environment variables and local variables is environment variables are passed to child processes while local variables are not:

Local Variable:

```
nsavva@maxwell:~$ x=3
nsavva@maxwell:~$ echo $x
3
nsavva@maxwell:~$ bash
nsavva@maxwell:~$ echo $x

nsavva@maxwell:~$
```

Environment Variable:

```
nsavva@maxwell:~$ export x=myvalue
nsavva@maxwell:~$ echo $x
myvalue
nsavva@maxwell:~$ bash
nsavva@maxwell:~$ echo $x
myvalue

nsavva@maxwell:~$
```

Environment Variables Again...

When we say the Shell passes environment variables to its child processes, we mean a copy is passed. If the variable is changed in the child process it is **not** changed for the parent

Example:

```
nsavva@maxwell:~$ export x=value1
nsavva@maxwell:~$ bash
nsavva@maxwell:~$ echo $x
value1
nsavva@maxwell:~$ export x=value2
nsavva@maxwell:~$ exit
nsavva@maxwell:~$ echo $x
value1
```

Listing and Removing Variables

- `env` - displays all environment variables
- `set` - displays all shell/local variables
- `unset name` - remove a shell variable
- `unsetenv name` - remove an environment variable

Shell Expansions

The shell interprets $ in a special way.

- If var is a variable, then `$var` is the value stored in the variable var.
- If `cmd` is a command, then `$(cmd)` is translated to the result of the command `cmd`.

Example

```
nsavva@maxwell:~$ echo $USER
nsavva
nsavva@maxwell:~$ echo $(pwd)
/home/nsavva
```

Arithmetic Expansion

The shell will expand arithmetic expressions that are encased in $((expression))

Examples

```
nsavva@maxwell:~$ echo $((2+3))
5
nsavva@maxwell:~$ echo $((2 < 3))
1
nsavva@maxwell:~$ echo $((x++))
3
```

And many more.

Note: the post-increment by 1 operation (++) only works on variables

Quotes

3 different types of quotes, and they have different meanings:

- Single quotes (’): Enclosing characters in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.
- Double quotes (“): Enclosing characters in double quotes preserves the literal value of all characters within the quotes, with the exception of $ ’ \ !
- Back quotes (`): Executes the command within the quotes. Like $().

Quotes

Example

```
nsavva@maxwell:~$ echo "$USER owes me $ 1.00"
nsavva owes me $ 1.00

nsavva@maxwell:~$ echo '$USER owes me $ 1.00'
$USER owes me $ 1.00

nsavva@maxwell:~$ echo "I am $USER and today is `date`"
I am nsavva and today is Mon Feb 09 11:30:42 EST 2015
```

Running Commands Sequentially

The ; Operator

`<command1> ; <command2>`

- Immediately after command1 completes, execute command2

The && Operator

`<command1> && <command2>`

- command2 executes **only if** command1 executes successfully

Example:

```
mkdir photos && mv *.jpg photos/
```

- Creates a directory and moves all jpegs into it

Exit Codes

The command after a && only executes if the first command is successful, so how does the Shell know?

- When a command exits it always sends the shell an exit code (number between 0 and 255)
- The exit code is stored in the variable $?
- An exit code of 0 means the command succeeded
- The man page for each command tells you precisely what exit codes can be returned

Example:

```
nsavva@maxwell:~$ ls ~/Documents/cs2043
2012 2013 2014 2015
nsavva@maxwell:~$ echo $?
0
```

You have the power!

We now have a variety of UNIX utilities at our disposal and it is time to learn about

scripting!

Scripting 101

Definition:

A script is very similar to a program, although it is usually much simpler to write and it is executed from source code (or byte code) via an interpreter. *Shell scripts* are scripts designed to run within a command shell like `bash`.

Scripts are written in a scripting language, like perl, ruby, python, sed or awk. They are then run using an interpreter. In our case, the scripting language and the interpreter are both **bash**.

The Shebang

All the shell scripts we'll see in this course begin the same way: with a **shebang** (#!). This is followed by the full path of the shell we'd like to use as an interpreter: `/bin/bash`

Example:

```
#!  /bin/bash
# This is the beginning of a shell script.
```

- Any line that begins with # (except the shebang) is a comment
- Comments are ignored during execution - they serve only to make your code more readable.

Simple Examples:

Bash scripts can be as simple as writing commands in a file.

Example: hello.sh

```
#!  /bin/bash
echo "Hello World"
```

Now set your file permissions to allow execution:

Example:

```
chmod u+x hello.sh
```

And finally you can run your first shell script!

```
./hello.sh
Hello World!
```

Hello World - String Version

Lets modify this slightly and use a variable:

Example: hello2.sh

```
#!  /bin/bash
STRING="Hello again, world!"
echo $STRING
```

Set your permissions and run:

```
chmod u+x hello2.sh && ./hello2.sh
Hello again, world!
```

A Backup Script

Here is something a little more practical - a simple script to back up all the files in your documents directory:

Example: backup.sh

```
#!  /bin/bash
tar -czf ~/backups/cs2043backup.tar.gz ~/cs2043/
```

Backup Script With Date

Lets add the current date to the name of our backup file.

Example: backupwithdate.sh

```
#!  /bin/bash
tar -czf ~/backups/cs2043_$(date +%d_%m_%y).tar.gz ~/cs2043/
```

Today, this will write to a file named `cs2043_09_02_2015.tar.gz`

Passing arguments to scripts

When we pass arguments to a bash script, we can access them in a very simple way:

- `$1, $2, ... $10, $11` : are the values of the first, second etc arguments
- `$0` : The name of the script
- `$#` : The number of arguments
- `$*` : All the arguments, "$*" expands to "$1 $2 ... $n",
- `$@` : All the arguments, "$@" expands to "$1" "$2" ... "$n"
- You almost always want to use `$@`
- `$?` : Exit code of the last program executed
- `$$` : current process id.

Simple Examples

multi.sh

```
#!  /bin/bash
echo $(( $1 * $2 ))
```

- Usage: ./multi.sh 5 10
- Returns first argument multiplied by second argument
- To do arithmetic in bash use $((math))

uptolow.sh

```
#!  /bin/bash
tr '[A-Z]' '[a-z]' < $1 >  $2
```

- Usage: `./uptolow.sh file1 file1low`
- translates all upper case letters to lowercase and writes to file1low

If conditionals

If statements are structured just as you would expect:

```
if cmd1
then
     cmd2
     cmd3
elif cmd4
then
     cmd5
else
     cmd6

fi
```

- Each conditional statement evaluates as true if the `cmd` executes successfully (returns an exit code of 0)

A simple script

textsearch.sh

```
#!  /bin/bash
# This script searches a file for some text then
# tells the user if it is found or not.
# If it is not found, the text is appended
if grep "$1" $2 > /dev/null
then
     echo "$1 found in file $2"
else
     echo "$1 not found in file $2, appending."
     echo $1 >> $2

fi
```

test expressions

We would not get very far if all we could do was test with exit codes. Fortunately bash has a special set of commands of the form `[ testexp ]` that perform the test **testexp**. First to compare two numbers:

- `n1 -eq n2` : tests if $n1 = n2$
- `n1 -ne n2` : tests if $n1 \neq n2$
- `n1 -lt n2` : tests if $n1 < n2$
- `n1 -le n2` : tests if $n1 \leq n2$
- `n1 -gt n2` : tests if $n1 > n2$
- `n1 -ge n2` : tests if $n1 \geq n2$

If either $n1$ or $n2$ is not a number, the test fails.

Test Expressions

We can use test expressions in two ways:

- `test EXPRESSION`
- `[ EXPRESSION ]`

Either of these commands returns an exit status of 0 if the condition is true, or 1 if it is false.

Use `man test` to learn more about testing expressions

Note: Remember you can check the exit status of the last program using the $? variable.

Example

```
#!  /bin/bash
# Created on [2/20/2009] by David Slater
# Purpose of Script:  Searches a file for two strings and prints which
#is more frequent
# Usage:  ./ifeq.sh <file> string1 string2

arg=‘grep $2 $1 | wc -l‘
arg2=‘grep $3 $1 | wc -l‘
if [ $arg -lt $arg2 ]
then
     echo "$3 is more frequent"
elif [ $arg -eq $arg2 ]
then
     echo "Equally frequent"
else
     echo "$2 is more frequent"

fi
```

string comparison

To perform tests on strings use

- `s1 == s2` : s1 and s2 are identical
- `s1 != s2` : s1 and s2 are different
- `s1` : s1 is not the null string

Make sure you you leave spaces! `s1==s2` will fail!

Expansion

When using `testexp` variable substitution is performed, but no matching is perform.

If `x` is the null string, what will `[ $x != monster ]` return?

Expansion

When using `testexp` variable substitution is performed, but no matching is perform.

If `x` is the null string, what will `[ $x != monster ]` return?

It will return an error, because $x is expanded to the null string and the test becomes `[ != monster ]` . To make sure there are no errors, place your variables inside double quotes. Then `[ $x != monster ]` is expanded to `[ "" != monster ]` which returns true.

path testing

If **path** is a string indicating a path, we can test if it is a valid path, the type of file it represents and the type of permissions associated with it:

- `-e path` : tests if **path** exists
- `-f path` : tests if **path** is a file
- `-d path` : tests if **path** is a directory
- `-r path` : tests if you have permission to read the file
- `-w path` : tests if you have write permission
- `-x path` : tests if you have execute permission

More stuff

We can now begin to ensure our scripts get the input we want:

```
if [ -f $1 ]
then
    Perform the action you want
else
    echo "This script needs a file as its input
    dummy!"
fi
```

More on testing

You can combine tests:

```
if [ testexp1 -a testexp2 ]
then
    cmd
fi
```

- `-a` : and
- `-o` : or
- `!  testexp1` : not

A note about debugging

To debug your code, invoke the script with the `-x` option. You will then see all the commands successfully executed:

```
$ bash -x ifeq.sh frankenstein.txt monster the
++ grep monster frankenstein.txt
++ wc -l
+ arg=33
++ grep the frankenstein.xt
++ wc -l
+ arg2=3850
+'[' 33 -lt 3850 ']'
+ echo 'the is more frequent'
```

Putting it on one line

Sometimes we might want to type a multiline command into the shell, we can do this by hitting enter for each line, or by using semicolons to tell the shell to start new lines:

Example:

```
if [ testexpr ] ; then command1 ; command2 ; fi
```

Real Example:

```
if [ $?  -eq 0 ] ; then echo "Last Command Successful" ; fi
```

Putting it on multiple line

Sometimes we might want to type a multiline command into the shell, we can do this by hitting enter for each line, or by using semicolons to tell the shell to start new lines:

Example:

```
if [ testexpr ]
then
partofcommand1 \
command1continued
command2;
fi
```

Real Example:

```
if [ $?  -eq 0 ]
then
ls \
*.txt
pwd
fi
```

Next Time