CS2043 - Unix Tools & Scripting
Lecture 10
Shell Scripting II

Spring 2015 [1]

Instructor: Nicolas Savva

February 11, 2015

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

Announcements

If conditionals (review)

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)

Exit Codes (review)

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
```

Test Expressions (review)

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.

Arithmetic Expansion (review)

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

Arithmetic

A little arithmetic can be useful and BASH can perform all the standard operators

Arithmetic

- a++, a- - : Post-increment/decrement
- ++a, - -a : Pre-increment/decrement
- a+b, a-b : Addition/subtraction
- a*b, a/b : Multiplication/division
- a%b : Modulu
- a**b : Exponential
- a>b, a<b : Greater than, less than
- a==b, a!=b : Equality/inequality
- =, +=, -= : Assignments

Using Arithmetic Expressions

We have already seen one way to do arithmetic:

Example:

```
echo $((2+5))
7
```

We can also use it as part of a larger command:

The "Let" Built-In

```
VAR1=2
let VAR2=$VAR1+15
let VAR2++
echo $VAR2
18
```

- `let` evaluates all expressions following the equal sign

The Difference

There are two major differences:

- all characters between the ((and)) are treated as quoted (no shell expansion)
- The let statement requires there be no spaces **anywhere** (so need to quote)

Example:

```
let "i=i + 1"
i=$(($i + 1))
```

The `while` loop

```
while cmd
do
    cmd1
    cmd2
done
```

Executes `cmd1, cmd2` as long as `cmd` is successful (i.e. its exit code is 0).

While loop example

```
i="1"
while [ $i -le 10 ]
do
    echo "$i"
    i=$(($i+1))
done
```

This loop prints all numbers 1 to 10.

Until loop

```
until cmd
do
    cmd1
    cmd2
done
```

Executes `cmd1, cmd2` as long as `cmd` is unsuccessful (i.e. its exit code is not 0).

Until loop example

```
i="1"
until [ $i -ge 11 ]
do
    echo i is $i
    i=$(($i+1))
done
```

for loop

```
for var in string1 string2 ...  stringn
do
    cmd1
    cmd2
done
```

The for loop actually has a variety of syntax it can accept. We will look at each in turn.

for loop example

```
#!  /bin/bash
# lcountgood.sh
i="0"
for f in "$@"
do
     j=‘wc -l < $f‘
     i=$(($i+$j))
done
echo $i
```

Recall that $@ expands to all arguments individually quoted ("arg1" "arg2" etc).

This script counts lines in a collection of files. For instance to count the number of lines of all the files in your current directory just run `./lcountgood.sh *`

for loop example

What happens if we change $@ to $*? Recall that $* expands to all arguments quoted together ("arg1 arg2 arg3")

```
#!  /bin/bash
# lcountbad.sh
i="0"
for f in "$*"
do
      j=‘wc -l < $f‘
      i=$(($i+$j))
done
echo $i
```

This does not work! Lets look at why.

Why we don't like $*

Consider

```
#!  /bin/bash
# explaingood.sh
j=0
for i in "$@"
do
j=$(($j+1))
echo $i
done

echo $j
```

This simply echos all the files you pass to the script and how many.

```
$ ./explaingood.sh *
explainbad.sh
explaingood.sh
lcountright.sh
lcountwrong.sh

4
```

Why we don't like $*

But if we change to $*

```
#!  /bin/bash
# explainbad.sh
j=0
for i in "$*"
do
j=$(($j+1))
echo $i
done

echo $j
```

This simply echos all the files at once and the number 1:

```
$ ./explaingood.sh *
explainbad.sh explaingood.sh lcountright.sh lcountwrong.sh

1
```

other for loop syntax

We can also do things like:

```
for i in {1..10}
do
    echo $i
done
```

To print 1 to 10.

other for loop syntax

We can also do things like:

```
for i in $(seq 1 2 20)
do
     echo $i

done
```

```
1
3
5
7
9
11
13
15
17
19
```

even **more** for loop syntax!

We can also do something more traditional:

```
for (( c=1; c<=5; c++))
do
     echo $c

done
```

To print 1 to 5 (spaces around c=1 etc do not matter)

An infinite loop

We can now create infinite for loops if we want

```
for (( ; ; ))
do
     echo "infinite loop [hit CTRL+C to stop]"

done
```

can't catch a break

We can use `break` to exit `for`, `while` and `until` loops early

```
for i in some set
do
      cmd1
      cmd2
      if (disaster-condition)
      then
            break
      fi
      cmd3
done
```

continue

We can use `continue` to skip to the next iteration of a `for`, `while` or `until` loop.

```
for i in some set
do
      cmd1
      cmd2
      if (i don't like cmd3-condition)
            continue
      fi
      cmd3
done
```

Reading in input from the user

You can ask the user for input by using the `read` command

read

`read varname`

- Asks the user for input
- By default stores the input in $REPLY
- Can read in multiple variables `read x y z`
- -p option allows you to print some text

Example:

```
read -p "How many apples do you have?  " apples
How many apples do you have?  5
$ echo $apples
5
```

Other uses for read

`read` can also be used to go line by line through a file or any other kind of input:

Example:

```
cat /etc/passwd | while read LINE ; do echo $LINE done
```

- Prints the contents of /etc/passwd line by line

```
ls *.txt | while read LINE ; do newname=$(echo $LINE |\
sed 's/txt/text/' ); mv -v "$LINE" "$(newname)" ; done
```

- Renames all .txt files in the current directory as .text files.

case

`case` allows you to execute a sequence of if else if statements in a more concise way:

```
case expression in

     pattern1 )
          statements ;;
     pattern2 )
          statements ;;
     ...

esac
```

Here the patterns are expanded using **shell expansion**. We can use match one of several patterns by separated by a pipe `|`.

superficial example

```
$ type=short
$ case $type in
tall)
echo "yay tall"
;;
short | petite)
echo "your height is either short or petite"
;;
hid*)
echo "variable type starts with hid..."
;;
*)
echo "none of the cases matched :("
;;
esac
```

```
your height is most likely not that great
```

- the case statement stops the first time a pattern is matched
- the case *) is a catchall for whatever did not match.

Next Time