CS265
Advanced Programming Techniques

Bash Scripting – Part 2

Lecture continuing from last week…

Topics covered last week

- Bash as a scripting language
- How to write a simple script
- How to execute it
- Shell Script Variables
    1. Command-line arguments
    2. Process-related variables
    3. Environment variables
    4. Shell variables
    5. User-defined variables

Topics covered this week

- Control structures

Control Structures

Control Structures

- We have branching:

```
if

if-else

if-elif-else

case
```

- And loops:

```
while

until

for

select
```

- And they all needs conditions / tests

Conditions / Tests

There are many and often confusing ways to test for a condition

1. `test, [ ]`	Provides string, numeric, and file tests
2. `[[ ]]`	Similar to [ ], but gentler syntax
3. `let, (( ))`	Provides numeric tests and arithmetic

1. Test a Condition - Using `test`

1. General form

```
test expr
```

2. Example

```
test $name = “Dimitra”
```

Spaces here are important to the left and right of =

```
if test $name = “Dimitra”
then
        echo 'Hello Dimitra!’
fi
```

`test` and `[]` behave similarly

1. Test a condition - Using [ ]

1. General form

```
[ expr ]
```

2. Example

```
[ $name = "Dimitra" ]
```

Spaces are important to the left and right of =

Spaces are important to the right of [and to the left of]

```
if [ $name = "Dimitra" ]
then
        echo 'Hello Dimitra!'
fi
```

Warning – Be careful with those spaces!

These are ok

```
[ $a = $b ]
```

```
[ $a=$b ]
```

These are not ok

```
[ $a = $b]
```
(common mistake)

```
[$a = $b ]
```

```
[ $a =$b ]
```

```
[ $a= $b ]
```

```
[$a=$b]
```

```
[$a = $b]
```

[ ] – Tests

- Built into Bash
  - But behaves like the disk utility (less than pretty)
  - Only such test available in Bourne shell
- Note, the spaces around the [ ] are necessary

Supports

- String tests
- File tests
- Numeric tests
- Logical operators tests

Mostly for stings and files!

[ ] – String tests

We have the normal binary, relational operators for strings

```
< = != >
```

e.g,

```
[ $var = "Dimitra" ]
```

```
[ $var < "Dimitra" ]
```

oops! < is a shell metacharacter

```
[ $var \< "Dimitra" ]
```

[ ] – String tests (continued)

[ ] supports unary tests for strings

`-z` True if string is empty

`-n` True if string is not empty

e.g,

```
[ -z ”$1” ]
```

Checks to see if the first argument is empty

[ ] - File tests

`-a file`	True if `file` exists
`-e file`	True if `file` exists
`-f file`	True if `file` is an regular file
`-c file`	True if `file` is an character file
`-d file`	True if `file` is a directory
`! -d file`	True if `file` is not a directory

```
if [ -e $file ]
then
    echo "file exists";
fi
```

[ ] - File Tests (continued)

There are some binary operators for files:

`f1 -nt f2` True if `f1` is newer than `f2`

`f1 -ot f2` True if `f1` is older than `f2`

`f1 -ef f2` True if `f1` is a hard link to `f2` (they are the same file)

e.g.,

`[ -e "$log" ]` to see if the log file exists

`[ -r "$input"]` to see if the input file is readable

`[ "$input" -nt "$log" ]` to see if the input file is newer that the log file

[ ] - Numeric Tests

Bash condition	Java Condition	Python Condition
n1 –eq n2	n1 == n2	n1 == n2
n1 –lt n2	n1 < n2	n1 < n2
n1 –gt n2	n1 > n2	n1 > n2
n1 –ne n2	n1 != n2	n1 != n2
n1 –le n2	n1 <= n2	n1 <= n2
n1 –ge n2	n1 >= n2	n1 >= n2

```
if [ $a -lt $b ]
then
    echo ”I found a to be less that b";
fi
```

NOTE:
you can’t use <,> inside [ ]
to compare numbers

[ ] – Numeric tests - WARNING

Use this

```
[ 1 -lt 2 ]
```

Not this

```
[1 -lt 2 ]
```

```
[ 1 -lt 2]
```

```
[ 1 < 2 ] - error
```

```
[ 13 < 2 ] - error
```

`[ 1 \< 2 ]` (this is syntactically ok but it compares strings TRUE)

`[ 13 \< 2 ]` (this is syntactically ok but it compares strings TRUE)

`[ 43 \< 2 ]` (this is syntactically ok but it compares strings FALSE)

[ ] – Logical Operators Tests

`! expr`

NOT - True when `expr` is false, false otherwise

`exp1 -a exp2`

AND - True when both `exp1` and `exp2` are true, false otherwise

`exp1 -o exp2`

OR - False when both `exp1` and `exp2` are false, true otherwise

2. Test a Condition - Using [[ ]]

- Is a built-in, so, syntax is gentler
  - Shell metacharacters `<` , `>` , etc., don't need to be escaped
  - Shell knows it's in a test
- Supports all the same tests as `[ ]`
- Supports the familiar logical operators:

  `! && ||`
- Tests using == and = are equivalent
- `==` `!=` treat the right operand as a pattern (glob / wildcard matching)

```
[[ abcde.f == a*e.? ]]
```

- `=~` treats the right operand as an extended regular expression (egrep)

```
[[ abcde.f =~ a.*e\..? ]]
```

Again don't use [[ ]] for arithmetic tests using >, <, ..

Similar to [ ], [[ ]] does string comparisons

3. Test a Condition - Using (( ))

- Treats values in double parameters as integers
- Performs arithmetic test
- Spaces don’t matter

```
x=5
if (( $x > 7 ))
then
    echo ”$x is greater than 7";
fi
```

The `let` command

1. Using the `let` command

```
let arithmetic-expr
```

2. Example

```
let "X = 1 + 1"

let "X=X + 1"

let "X = $X+1"

echo $X
```

Spaces here are not important!

(( )) and let - arithmetic operators

`(( ))` and `let` can be used to evaluate arithmetic exceptions

- Arithmetic: `** * / % + -`
- Bit-wise: `~ << >> ^ & |`
- Pre- and post-fix increment/decrement: `++ --`
- A Java-like ternary operator: `?:`
- Assignment `(=)`, and the usual operator/assignment operators: `+= -= &=`, etc.

(( )) - examples

```
$ x=13
$ echo $(( x+15 ))
28
$ echo $x
13
$ (( y = x*4 ))
$ echo $y
52
$ (( y-=1 ))
$ echo $y
51
$ echo $((x>>2))
3
```

Variables – what is the difference?

`let a=1+1`	variable creation with simple arithmetic expansion
`a=$((1+1))`	arithmetic expansion, simple arithmetic; the result of the expression replaces the expression
`a=$[1+1]`	old syntax, don't use
`a=$(expr 1 + 1 )`	calls the Unix `expr` command
`a=$(ls)`	assigns the output of `ls` to `a`

Let’s see the control structures now…

if

```
if tests
then
        cmds;
fi
```

Also

```
if tests; then cmds; fi
```

if

`if tests; then cmds; fi`

- `tests` is executed
- If the exit status is 0 (success), `cmds` is executed

```
if grep Waldo * &> /dev/null ; then # std out/err go to bit bucket
echo "Found Waldo!"
fi
```

```
if [[ -d "$paris" && -r "$paris" ]] ; then
echo "I see $paris"'!'
fi
```

```
if (( $cats > 3 )) ; then
echo "Too many cats"
echo "People will talk"
fi
```

if-else

```
if tests
then
        cmds;
else
        cmds;
fi
```

if-else

```
if tests; then cmds; else cmds; fi
```

```
if grep Waldo * &> /dev/null ; then
  echo "Found Waldo!"
else
  echo "Waldo's a slippery one"
fi
```

```
if [[ -d "$paris" && -r "$paris" ]] ; then
  echo "I see $paris"'!'
else
  echo "Might be on the wrong continent"
fi
```

```
if (( $cats > 3 )) ; then
  echo "Too many cats"
  echo "People will talk"
else
  echo "You might yet be sane"
fi
```

if-elif-else

```
if tests
then
        cmds;
elif tests
        cmds;
…
else
        cmds;
fi
```

if-elif-else

```
if tests; then cmds; elif cmds; else cmds; fi
```

```
read grade
if (( $grade >= 90 )) ; then
echo "A"
elif (( $grade >= 80 )) ; then
echo "B"
elif (( $grade >= 70 )) ; then
echo "C"
elif (( $grade >= 60 )) ; then
echo "D"
else
echo "F"
fi
```

case

```
case word in
pattern1) cmds;;
pattern2 | pattern3) cmds;;
…
patternN) cmds;;
'*') cmds;; #otherwise
esac
```

Patterns can contain wildcards

It uses patterns not regular expressions

case

`case word in {pattern ) cmds ;;} esac`

- Selectively execute `cmds` if `word` matches the corresponding `pattern`
- Commands are separated by ;
- Cases are separated by ;;

```
#!/bin/bash

case $1 in
  n ) echo “n” ;;
  x ) echo “x” ;;
  \? | h | H ) echo "Use option n or option x" ; exit 1 ;;
  ?) echo "Unknown character" ;;
esac
```

`while` Loop

```
while condition
do
        command(s)
done
```

while Loop

`while tests; do cmds; done`

- `tests` is executed
- If the exit status is 0 (success), `cmds` is executed
- Execution returns back to `tests` , start again

```
i=0
while (( $i<=12 )) ; do
echo $i
(( i+=1 ))
done
```

```
cat list | while read f ; do
# Assume list contains one filename per line
stat "$f"
done
```

`until` Loops

```
until condition
do
        command(s)
done
```

`for` Loops

```
for variable in list
do
        command(s)
done
```

- `variable` is a user variable
- `list` is a sequence of strings separated by spaces

for loop

```
for name [in list]; do cmds ; done
```

- Executes `cmds` for each member in `list`
- `"$@"` used if list isn't there

```
for i in a b c ; do
> echo $i
> done
a
b
c
```

```
for id in $(cat userlist) ; do
# assumes no spaces in userIDs
echo "Mailing $id..."
mail -s "Good subject" "$id"@someschool.edu < msg
done
```

for loop

- bash has a C/Java-like for loop:

```
% for (( i=0; i<3; ++i )) ; do
> echo $i
> done
0
1
2
```

```
% for (( i=12; i>0; i-=4 )) ; do
> echo $i
> done
12
8
4
```

select

```
select name [in list]; do
        cmds ;
done
```

select

`select name [in list]; do cmds ; done`

- Much like the `for` loop
- Displays enumerated menu of `list`
- Puts user's choice in `name`

```
select response in "This" "That" "Quit"
do
   echo "You chose $response"
done
```

`break` and `continue`

- Interrupt loops (`for`, `while`, `until`)
- `break` jumps to the statement after the nearest done statement
  - terminates execution of the current loop
- `continue` jumps to the nearest done statement
  - brings execution back to the top of the loop

Loops – continue, break

- `break` exits a loop
- `continue` short-circuits the loop, resumes at the next iteration of the loop

```
for i in {1..42} ; do
   (( i%2 == 0 )) && continue
   (( i%9 == 0 )) && break
   echo $i
done
```

```
1
3
5
7
```

User Input

1. `read` command
2. `printf` command
3. Using command line arguments

1. Reading User Input – `read` command

- Syntax: `read varname`
  - No dollar sign
- Reads from standard input
- Waits for the user to enter something followed by `<RETURN>`
- Stores what is read in user variable
- Can be used with a special prompt

```
read –p prompt variable
```

- To use the input: `echo $varname`
  - Needs dollar sign

1. Reading User Input – `read` command

- More than one variable may be specified

```
read var1 var2 var3
```

- Each word will be stored in separate variable
- If not enough variables for words, the last variable stores the rest of the line

2. Generating Output – `printf` command

- The command syntax

```
printf format [arguments ...]
```

- prints its optional `arguments` under the control of the `format`, a string which contains three types of objects:
  - plain characters, which are simply copied to standard output
  - character escape sequences which are converted and copied to the standard output, e.g., \t (tab), \n (newline), \a (bell), etc.
  - and format specifications, each of which causes printing of the next successive `argument`, e.g., \s (string), \c (character), etc.

```
$ printf "hi "
$ printf "hi\n"
$ printf "hi\a\a\a\a"
$ printf "hi %s\n" "Hello"
```

```
Prints hi without a newline
Prints hi and then a newline
Prints hi and rings the bell 3 times
Prints hi Hello and a newline
```

3. Reading user input – using command-line arguments

- $1, $2, … normally store command line arguments
- Their values can be changed using the `set` command

```
set newarg1 newarg2 …
```

NOTE:
This is more important than it looks

All command-line arguments

- Both `$@` and `$*` get substituted by all the command line arguments
- They are different when double-quoted
- `"$@"` expands such that each argument is quoted as a separate string
- `"$*"` expands such that all arguments are quoted as a single string

Reading user Input - Quoting Issues

- What if I want to output a dollar sign?
- Two ways to prevent variable substitution:

```
echo '$dir’
echo \$dir
```

- Note: echo `"$dir"` is the same as echo `$dir`

Back Quotes

Enclosing a command invocation in back quotes (the character usually to the left of 1) results in the whole invocation substituted by the output of the command

```
dateVar=`date`

echo $dateVar

Mon 16 Sep 2019 10:29:26 EDT
```

Better Code instead of using back quotes:

```
dateVar=$(date)
echo $dateVar
```

Arithmetic Operations Using `expr`

- The shell is not intended for numerical work
- However, the `expr` utility may be used for simple arithmetic operations on integers

```
sum=`expr $1 + $2`
```

- Note: spaces are required around the operator + (but not allowed around the equal sign)

Shell Script Functions

- Syntax:

```
function_name()
{
        command(s)
}
```

- Allows for structured shell scripts

`bash` functions

```
function name {body}

function name() {body}
```

- Executed in the same environment
- Arguments to function are handled the same as arguments to a script
- Can be called recursively
- Built-in `return rv` can be used in a function, to return execution (and optional status `rv`) to caller

Function Examples

```
function hello
{
echo "hello $1"
if [[ -n "$2" && "$2" -gt 1 ]] ; then
hello $1 $(($2-1))
fi
}
```

Called as a script from the shell prompt the output will be

```
$ hello Dimitra 3
hello Dimitra
hello Dimitra
hello Dimitra
```

Function Examples - Continued

```
usage()
{
   echo "demo: usage: demo arg
}

while [ $# -eq 0 ]
do
   usage ; exit
done

echo "Continue with program..."
```

Sample output

```
$ demo
demo: usage: demo arg
$ demo arg
Continue with program...
```

{x..y} – Brace Expansion

```
{x..y}
```

- Generates sequences in a natural way

```
echo {5..13} # no spaces around the { and } symbols
5 6 7 8 9 10 11 12 13
```

```
echo {a..g}
a b c d e f g
```

- Brace expansion will pad numbers on the left

```
for i in {0..5} ; do echo -n "$i " ; done
0 1 2 3 4 5
```

- This is quite handy in loops:

```
for i in {0..5} ; do
\rm proc${i}.log
done
```

Local variables in functions

`local {var}`

- Defines variable(s) local to function
- Won't step on caller's environment

```
function hello {
    local USER='Elmer Fudd'
    FOO='Hunting Wabbit'
    echo "Hello, $USER, you are $FOO"
}
```

```
$ FOO='Baking Cookies'
$ echo $USER
dv35
$ hello
Hello, Elmer Fudd, you are Hunting Wabbit
$ echo $FOO
Hunting Wabbit
$ echo $USER
dv35
```

Command Output

- To retrieve the output of a bash command

```
output=$( ./my_script.sh )
```

- e.g.,

```
output=$(ls -l)

echo $output
```

Bash Arrays

- Arrays can contain both numbers and strings

```
myArray=(1 2 "three" 4 "five")
```

- Use curly braces {} to access the array elements

```
${myArray[0]} ${myArray[1]} ${myArray[2]}
```

- All the elements in the array

```
myArray[@]
```

- Loop through array elements

```
 for t in ${myArray[@]}; do
        …

done
```

- Loop through array indices

```
 for i in ${!myArray[@]}; do
        …
done
```

Bash Arrays - continued

`arr=()`	Create an empty array
`arr=(1 2 3)`	Initialize array
`${arr[2]}`	Retrieve third element
`${arr[@]}`	Retrieve all elements
`${!arr[@]}`	Retrieve array indices
`${#arr[@]}`	Calculate array size
`arr[0]=3`	Overwrite 1st element
`arr+=(4)`	Append value(s)
`str=$(ls)`	Save ls output as a string
`arr=( $(ls) )`	Save ls output as an array of files
`${arr[@]:s:n}`	Retrieve n elements starting at index s

Syntactic nightmare?

- Not the most intuitive of syntax perhaps

- E.g., add an new val at the end of an array

```
a[${#a[@]}]=val
```

- And then display it

```
echo ${a[${#a[@]}-1]}
```

Lessons

- Lesson 1: Bash is a veteran IT workforce
- Lesson 2: Bash most useful for system administration, web app development, data crunching (when the job is to communicate and pipe through commands)
- Lesson 3: Learn bash and you will own your box

Resources

- These notes
- Unix in a nutshell, 4th Edition (2005) http://tinyurl.com/tahe47t
  - Chapters 3 and 4

- Try the examples and do the exercises from the following two tutorial
  Link to Bash Tutorial

- Bash arrays (nice introduction)
  https://opensource.com/article/18/5/you-dont-know-bash-intro-bash-arrays

Acknowledgements

These slides are copied or inspired by the work of the following courses and people:

- CS265 & CS571, Drexel University, Kurt Schmidt, Jeremy Johnson, Geoffrey Mainland, Spiros Mancoridis, Vera available at https://www.cs.drexel.edu/~kschmidt/CS265/
- Bil Tzerpos's class