Basic Bash Scripting

Katherine Holcomb
UVACSE

Uses for Scripting

- Automate repetitive tasks
  - Could also use languages like Perl or Python but it's still handy to know some bash
- Write job scripts for the PBS queueing system
  - For most users these must be a shell script (bash, ksh, or tcsh)
- Anything you can type on the command line can be put into a script

Example

```
#!/bin/bash
#This is a bash script
echo “Hello world!”
echo “I am done now!” #my comment
```

- The first line tells the system to start the shell and execute the commands. It is sometimes called a “shebang.”

Executing Your Script

- Suppose the previous script was in a file `hello.sh`
- By default it is just text. You must explicitly make it executable

```
chmod a+x hello.sh
```

- Then you can just invoke it by name at the command line

```
./hello.sh
```

Comments and Continuations

- All text from the # to the end of the line is ignored
- To continue a statement on to the next line, type \<enter>

```
#This is a comment
echo “This line is too long I \
want it to be on two lines”
```

Sourcing

- If you execute the script as a standalone script, it starts a new shell -- that is what the shebang does—if you do not use that you can type

```
bash myscript.sh
```

- Sometimes you just want to bundle up some commands that you repeat regularly and execute them within the *current* shell. To do this you should not include a shebang, and you must *source* the script

```
. <script> or  source <script>
```

Variables

- Like all programming languages, the shell permits you to define variables.
- Variables and environment variables are essentially the same thing. Environment variables are conventionally written with all capital letters and are used to set properties. Shell variables are used for assignments and such.
- Arrays are supported
- Scalar variables are referenced with $
- When assigning variables do not put spaces around the equals sign!

String Expressions and Variables

- Bash isn't very good at math so much of what we do with it involves strings.
- String expressions
  - Single quotes ('hard quotes'): the expression is evaluated literally (no substitutions)
  - Double quotes ("soft quotes"): the $, \ (backslash), and ` (backtick) are taken to be shell characters

    $ indicates a variable name follows

    \ escapes the following character

    `<command>` command is executed and its output streams are returned

Examples

```
echo `date`
echo ‘This is worth $2’
var=2
echo “This is worth $2”
echo “This is worth $var”
echo “This is worth \$$var”
```

Conditionals

- The shell has an `if else` construct:

```
if [[ <condition> ]]
then
    commands
elif
    commands
else
    more commands
fi
```

Conditions

- The conditions take different forms depending on what you want to compare.
- String comparison operators:
- = != < > -z -n
- equal, not equal, lexically before, lexically after (both these may need to be escaped), zero length, not zero length.
- Can use == but behaves differently in [] versus [[]]

Example: String Comparisons

```
if [[ $name == “Tom” ]]; then
  echo Tom, you are assigned to Room 12
elif [[ $name == “Harry” ]]; then
  echo Harry, please go to Room 3
elif [[ -z $name ]]; then
  echo You did not tell me your name
else
  echo Your name is $name
fi
```

Numeric Comparisons

- Numeric comparisons are possible:

```
-eq -ne -gt -ge -lt -le

if [[ $a -eq 2 ]]; then
    stuff
fi
```

Testing File Properties

- This is a very common occurrence in bash scripts.
- There are many operators for this. These are the most common:

`-e <file>` : file exists

`-f <file>` : file exists and is a regular file

`-s <file>` : file exists and has length > 0

`-d <name>` : exists and is a directory

Example

```
if [[ -d mydir ]]; then
  if [[ -f the_file ]]; then
    cp the_file mydir
  fi
else
  mkdir mydir
  echo “Created mydir”
fi
```

Other Conditional Operators

- `!` : not ; negates what follows
- `-a` : and ; for compound conditionals
  can use && with [[]]
- `-o` : or ; for compound conditionals
  can use || with [[]]

Case Statement

```
case expression in
   pattern1 )
      statements ;;
   pattern2 )
      statements ;;
   pattern3 )
      statements ;;
esac
```

Loops

- The bash `for` loop is a little different from the equivalent in C/C++/Fortran (but is similar to Perl or Python)

```
for variable in iterator
do
   commands
done
```

Examples

```
for i in 1 2 3 4 5 ; do
  echo “I am on step $i”
done

for i in {1..5}  #bash 3.0 and up
do
  echo “I am on step $i”
done
```

Three-Expression For

for ((EXPR1; EXPR2; EXPR3))
do
    statements
done

- Example:

```
for (( i=0; i<$IMAX,i++ )); do
      echo $name”.”$i
done
```

while loop

```
while [ condition ]
do
   command
   command
   command
done
```

One of the commands in the while loop *must* update the condition so that it eventually becomes false.

break

- If you need to exit a loop before its termination condition is reached you can use the break statement.

```
while [ condition ]
do
   if [ disaster ]; then
     break
   fi
   command
   command
done
```

continue

- To skip the commands for an iteration use continue

```
for i in iterator; do
   if [[ something ]]; then
      continue
   fi
   command
   command
done
```

Bash Arithmetic

- We said earlier that bash is bad at arithmetic, but some basic operations can be performed.
- Expressions to be evaluated numerically must be enclosed in *double parentheses*. It works only with *integers*.

```
x=$((4+20))
i=$(($i+1))
```

If you need more advanced math (even just fractions!) you must use `bc`.

bc

- If you really, really, really must do math in a bash script, most of the time you must use bc
- x=$(echo “3*8+$z” | bc)

Command-Line Arguments

- Many bash scripts need to read arguments from the command line. The arguments are indicated by special variables $0, $1, $2, etc.
- $0 is the name of the command itself
- The subsequent ones are the options
- To have a variable number of options, use the `shift` built-in.

Example

```
#!/bin/bash
USAGE="Usage:$0 dir1 dir2 dir3 ...dirN"
if [ "$#" == "0" ]; then
   echo "$USAGE"
   exit 1
fi
while [ $# -gt 0 ]; do
   echo “$1”
   shift
done
```

More Useful Example

- *while* plus *case*

```
while [ $# -gt 0 ]; do
      case “$1” in
   -v)  verbose=“on”;;
   -*)
      echo >&2 “USAGE: $0 [-v] [file]”
      exit 1;;
   *)  break;;   # default
   esac
   shift
done
```

Functions

```
function name() {
   statement
   statement
   statement
   return $VALUE
}
```

- the keyword function is optional in newer versions of bash. The parentheses are always left empty.
- Function definitions must precede any invocations.

Function Arguments

- Function arguments are passed in the *caller*. In the function they are treated like command-line options.

```
#!/bin/bash
function writeout {
    echo $1
}
writeout “Hello”
```

Variables

- Variables set in the function are global to the script!

```
#!/bin/bash
myvar="hello"

myfunc() {

    myvar="one two three"
    for x in $myvar
    do
        echo $x
    done
}

myfunc

echo $myvar $x
```

Making Local Variables

- We can use the keyword `local` to avoid clobbering our global variables.

```
#!bin/bash
myvar="hello"
myfunc() {
    local x
    local myvar="one two three"
    for x in $myvar ; do
        echo $x
    done
}
myfunc echo $myvar $x
```

Return Values

- Strictly speaking, a function returns only its exit status.
- The returned value must be an integer
- You can get the value with $?
- e.g.

```
myfunc $1 $2
result=$?
```

Arrays

- Array variables exist but have a somewhat unusual syntax.
- Arrays are zero based so the first index is 0
- Obtaining the value of an item in an array requires use of ${}

```
val=${arr[i]}
```

- `${arr[@]}` # All of the items in the array
- `${#arr[@]}`# Number of items in the array
- `${#arr[0]}` # Length of item zero

Example

```
my_arr=(1 2 3 4 5 6)
for num in ${my_arr[@]}; do
    echo $num
done
```

Herefiles

- A herefile or here document is a block of text that is dynamically generated when the script is run

```
CMD << Delimiter
  line
  line
Delimiter
```

Example

```
#!/bin/bash
# 'echo' is fine for printing single line messages,
#+ but somewhat problematic for for message blocks.
# A 'cat' here document overcomes this limitation.

cat <<End-of-message
---------------------------------
This is line 1 of the message.
This is line 2 of the message.
This is line 3 of the message.
This is line 4 of the message.
This is the last line of the message.
---------------------------------
End-of-message
```

Some Resources

- Linux Shell Scripting Tutorial:

  http://bash.cyberciti.biz/guide/Main_Page
- How-To:

  http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

- Advanced Scripting Tutorial

  http://tldp.org/LDP/abs/html/