Unix Scripting

Lecturer: Shahdad Shariatmadari

May 2020

Agenda

- Introduction to Shell Scripting
  - Categories of variables
  - Conditional Statements
  - Loops

Shell script : Shebang

- Most Linux shell and Perl / Python script starts with Shebang
  - *#!/bin/bash*
- The #! syntax used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems
  - It is nothing but the absolute path to the Bash interpreter.
  - This ensures that Bash will be used to interpret the script, even if it is executed under another shell
  - Use the which utility to find out path to use: **`which bash`**

Variables

There are three general categories of variables

- Environment Variables
  - Variables that have been assigned by the Linux OS.
  - This variables are easy to remember and are commonly used.
  - Some of these variables cannot be changed by the user.

- User-Defined Variables
  - Variables set within the shell script to be used within the script.
  - The user can set and change these variables for their own purpose.

- Positional Parameters
  - These variables can be used 2 ways:
    - Assigned **inside** the shell script by the set command
    - Assigned when issuing a shell script with arguments

Example: ./myShellScript.bash arg1 arg2 arg3

Variables

Environment Variables

- Environment variables are used by the shell and many of these variable have values already assigned to them.
- Environment variables are usually identified as UPPERCASE letters.
- The user can see the values assigned to these variables by issuing the set command without an argument.
- Some of these variables can be changed by the user, some are assigned by the system and cannot be changed.

Variables

Environment Variables

- Keyword shell variables can be used in the shell script with Unix commands to “customize” the script for the particular user.

- Examples:

  – **echo “Hi there, $USER”** # Displays current user’s name.

  – **echo $PWD** # Displays user’s current directory.

  – **mkdir $HOME/dir1** # Creates a directory called dir1 that is
  # contained in user’s home directory.

Variables

Positional Parameters

- Positional Parameters have the feature that if your shell script is run with arguments, those arguments can be used as variables within your running shell script!

- This makes your shell script work like a “real” Linux OS command that accepts arguments.

- You can use the set command to assign values to these read-only shell variables inside the script as well..

Variables

Positional Parameters

- Parameter Parameters range from $1 to $9. To access higher numbers (command arguments), you must contain number in braces (eg. ${10}, ${25}, etc…)

- $0 is <u>script name</u> or is <u>shell name</u> if $0 used from shell prompt.

- Positional parameters are assigned values two ways:
  - Using the set command within the shell script. For Example:

```
set one two three
echo "First: $1, Second: $2, Third: $3"
```

  - Using the set command within the shell script. For Example:

```
./myShellScript.bash one two three
# Can use $1, $2, $3 in script…
```

Variables

Positional Parameters

- The shift command is used to move the positional parameters (i.e. arguments) one position to the left.
- As a result, the leftmost positional parameter is lost.
- A number as an argument after the shift command indicates how many positions to the left to shift.

Eg.

```
set one two three
echo $1 $2 $3
one two three
shift
echo $1 $2 $3
two three
```

Variables: Special Parameters

Special Parameters

- There are some special symbols that can be used to represent positional parameter information and other useful information such as process ID, exit status, etc…

$#	**number of positional parameters**
$*	**All positional parameters**
$@	**All positional parameters (each contained in**
$?	**Exit Status of previous command**
$$	**Current process ID Number**
$!	**Previous background process ID Number**

Using Logic

The purpose of the if statement is to execute a command or commands based on a condition

The condition is evaluated by a test command, represented below by a pair of square brackets

```
if [ condition ]
then
   command(s)
fi
```

if Statement Example

Test with a condition
Notice the spaces after “[“ and before “]”

```
read password

if [  ”$password” = ”P@ssw0rd!”  ]
then
  echo ”BAD PASSWORD!”
fi
```

Activity

- Develop a script which ask user to enter a password and check whether the password is equal to “admin123”
  - Modify the program to allow user to enter both ADMIN123 or admin123 as password
    - Any way to allow user to enter any of these
      - aDmin123, ADmin123, adMIN123

The test Command

- The test command can be used in two ways:
  - As a pair of square brackets: [condition]
  - The test keyword: test condition
- The condition test can result in success (0) or failure (1), unless the negation "not" (!), is used
- The test can compare numbers, strings, and evaluate various file attributes
  - Use = and != to compare strings,
    for example: ["$name" = "Bob"]
  - Use -z and -n to check string length,
    for example: [! -z "$name"]
  - Use -gt, -lt, -eq, -ne, -le, -ge for number,
    for example: [”$salary” -gt 100000]

The Test Command

- Common file test operations include:
  - -e (file exists)
  - -d (file exists and is a directory)
  - -s (file exists and has a size greater than zero)
  - -w (file exists and write permission is granted)
- Check man test for more details

Activity: Try the following code in command prompt

- `x=9`
- `test $x -eq 9`
- `echo  $?`
- What is the output?
- Change x=10, and try the above code again. What is the output?

Using Loops

- A for loop is a very effective way to repeat the same command(s) for several arguments such as file names
  Syntax:

Variable "item" will hold one item from the list every time the loop iterates

- for item in list
  do
  command(s)
  done

List can be typed in explicitly or supplied by a command

What does the following code do?

```
#!/usr/bin/bash

value=33
if test $value -eq 34
then
  echo “OK”
else
  echo “DIFFERENT”
fi
```

Question: what does the following script do?

- ```
  value=34
  if test $value -gt 2
  then
    if test $((value % 2)) -eq 0
    then
      echo “even”
    fi
  fi
  ```
  ```

Activity 1

- Change the previous script to read a number from input, and displays whether it is an even number or odd number
-
```
value=34
if test $value -gt 2
then
  if test $((value % 2)) -eq 0
  then
    echo “even”
  fi
fi
```

Activity 2

- Change the previous script to accept a number as **argument**, and displays whether it is an even number or odd number
-
```
value=34
if test $value -gt 2
then
  if test $((value % 2)) -eq 0
  then
    echo “even”
  fi
fi
```

elif control-flow statement

- The elif statement is used to work like a nested if statement. It performs another test, and execute the command(s) if the result is true.
-
```
if test $mark -gt 50
  then
    echo "you pass"
  elif test $mark -eq 50
    then
      echo "you JUST passed"
  else
    echo "sorry, you failed"
fi
```

for loop: using range

- *for number in {start..end..step}*
  - `for number in {1..10}`
  - The curly brackets {} basically denotes a range, and the range, in this case, is 1 to 10 (the two dots separate the start and end of a range).
- To loop between 0 and 100 but only show every tenth number
  - `for number in {0..100..10}`

Example

```
for number in {0..100..10}
do
echo $number
done
```

A More Traditional Looking For Loop

- You can, however, write a for loop in a similar style to the C programming language
  - *for((initialization; condition;alteration))*
- Example

```
for ((number=1;number < 10;number++))
do
  echo $number
done
```

While loop

- In most **computer programming** languages, a **while loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition.
- The while loop is used to repeat a section of code an unknown number of times until a specific condition is met.

While

- The second type of looping command to be described in this chapter is the while. The format of this command is
- `while condition`
  - `do`
    - `command`
    - `command ...`
  - `done`

Example

```
i=1
while [ "$i" -le 5 ]
do
   echo $i
   i=$((i + 1))
done
```

Activity: What does the following program do?

```
while [ "$#" -ne 0 ]
do
  echo "$1"
  shift
done
```

Activity: Explain what does the following scripts do?

```
for addr in $(cat ~/addresses)
do
        mail -s ”Newsletter” $addr < ~/spam/newsletter.txt
done
```

```
for count in 3 2 1 'BLAST OFF!!!'
do
        sleep 1
        echo $count
done
```

```
for id in $(seq 1 1000)
do
        mkdir student_$id
done
```