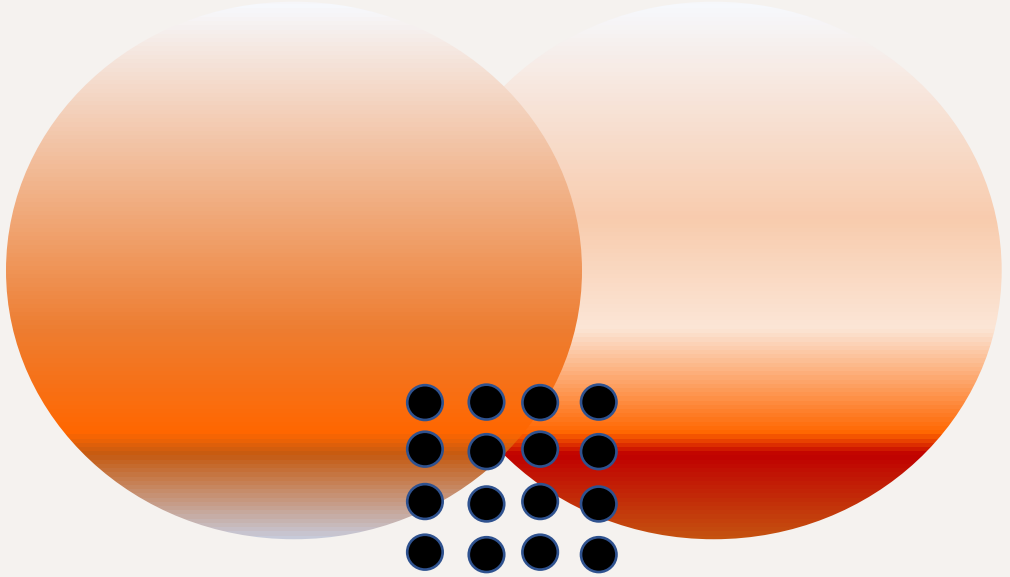
TECHNO INDIA SILIGURI INSTITUTE OF TECHNOLOGY
A Satyam Roychowdhury Initiative
NAAC ACCREDITED

UNIT 7: Shell Scripting

BCAN 601: UNIX and Shell Programming



Why write shell scripts?

- To avoid repetition:
  - If you do a sequence of steps with standard Unix commands over and over, why not do it all with just one command?
- To automate difficult tasks:
  - Many commands have subtle and difficult options that you don’t want to figure out or remember every time.

Introduction

- Basically, a shell script is a text file with Unix commands in it.
- Shell scripts usually begin with a #! and a shell name
  - For example: #!/bin/sh
  - If they do not, the user's current shell will be used
- Any Unix command can go in a shell script
  - Commands are executed in order or in the flow determined by control statements.
- Different shells have different control structures
  - The #! line is very important
  - We will write shell scripts with the Bourne shell (sh)

Introduction

- chmod the files to be executable; otherwise, you couldn't run the scripts
- $ $chmod\ u + x\ prog1.sh$
- Run them as normal commands:
- $./prog1.sh$

expr Computation and string handling

- expr command performs two type of operations:
  - Perform arithmetic operations on integers.
  - Manipulate strings

Assigning Command Output to a Variable

- Using backquotes, we can assign the output of a command to a variable:

```
#!/bin/sh
files=`ls`
echo $files
```

- Very useful in numerical computation:

```
#!/bin/sh
value=`expr 12345 + 54321`
echo $value
```

Using expr for Calculations

- Variables as arguments:

```
% count=5
% count=`expr $count + 1`
% echo $count
6
```

  - Variables are replaced with their values by the shell!
- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |
  - parentheses: (,)
  - precedence is the same as C, Java

Making Script Interactive

- The $read$ statement is the shell's internal tool for taking input from the user.
- $read\ var1$
- The script pause at this point and take input from keyboard and the entered value will store in the variable var1

The **read** Command (continued)

Read from stdin (screen)
Read until new line

Format	Meaning
`read answer`	Reads a line from `stdin` into the variable `answer`
`read first last`	Reads a line from `stdin` up to the whitespace, putting the first word in `first` and the rest of the of line into `last`
`read`	Reads a line from `stdin` and assigns it to `REPLY`
`read –a arrayname`	Reads a list of word into an array called `arrayname`
`read –p prompt`	Prints a prompt, waits for input and stores input in `REPLY`
`read –r line`	Allows the input to contain a backslash.

Positional Parameter

- Shell script can accept arguments from the command line itself. When arguments are specified with a shell script, they are assigned to certain special variables.
- The first argument is read by the shell into the parameter $1, the second argument into $2, and so on.

$1	The first argument
$2	The second argument
$0	The name of the script
$#	The number of argument
$*	The complete set of positional parameters as a string

Control Statements

- Without control statements, execution within a shell scripts flows from one statement to the next in succession.
- Control statements control the flow of execution in a programming language
- The three most common types of control statements:
  - conditionals: if/then/else, case, ...
  - loop statements: while, for, until, do, ...
  - branch statements: subroutine calls (good), goto (bad)

Test

Syntax for if is:

```
if [condition]
then
        statements/commands
fi
```

```
if [condition]
then
        #if block
        statements/command
else
        #else block
        statement/command
```

Test

```
if [ condition ]
then
        #if block
        Statements/commands
Elif [ condition ]
        # elif block
        Statements/commanda
else
        #else block
fi
```

```
read a
read b
if [ $a –lt $b ]
then
        echo a is less than b
elif [$a –gt $b ]
        echo a is greater than b
else
        echo a is equal to b
fi
```

File related test

Test Operator	Test True If
[*file1* **-nt** *file2* **]**	True if file1 is newer than file2*
[*file1* **-ot** *file2* **]**	True if file1 is older than file2*
[*file1* **-ef** *file2* **]**	True if file1 and file2 have the same device and inode numbers.
-b filename	Block special file
-c filename	Character special file
-d filename	Directory existence
-e filename	File existence

File related test

-f filename	Regular file existence and not a directory
-G filename	True if file exists and is owned nu the effective group id
-g filename	Set-group-ID is set
-k filename	Sticky bit is set
-L filename	File is a symbolic link
-p filename	File is a named pipe
-O filename	File exists and is owned by the effective user ID
-r filename	file is readable
-S filename	file is a socket
-u filename	Set-user-id bit is set
-w filename	File is writable
-x filename	File is executable

File related Test

```
#!/bin/sh
# Script to check permission
if [ -x $1 ]
then
echo file have read permission
 else
echo file does not have read permission
fi
```

Test command for string testing

Test Operator	Tests True if
`[ string1 = string2 ]`	String1 is equal to String2 (space surrounding = is necessary
`[ string1 != string2 ]`	String1 is not equal to String2 (space surrounding != is not necessary
`[ string ]`	String is not null.
`[ -z string ]`	Length of string is zero.
`[ -n string ]`	Length of string is nonzero.
`[ -l string ]`	Length of string (number of character)

Test command for string testing

```
#!/bin/bash
str1="data"
str2="server"
if [ "$str1" != "$str2" ];
then
        echo "The strings are different."
else
        echo "The strings are the same."
fi
```

The case conditional

```
Case expr in
Pattern1) command1
Command2 ;;
Pattern2) command3 ;;
……
*) command
esac
```

```
#!/bin/bash
tput clear
echo "\n 1. for find files for last 24
    hours\n 2. for disk space 3.space
    cpnsimed by this user 4. Exit\c"
read choice
case $choice in
1) find $HOME -mtime -1 -print ;;
2) df ;;
3)du -s $HOME ;;
4) exit;;
esac
```

The case conditional

```
#!/bin/bash
echo enter a single character.
read char
case $char in
[0-9]) echo you have entered a digit ;;
[a-z]) echo you have entered a lower case letter ;;
[A-Z] ) echo have entered a uppercase letter ;;
*) echo you have entered some special character
esac
```

while loop

- The while command evaluates the command following it and, if its exit status is 0, the commands in the body of the loop are exeeuted.
- The loop continues until the exit status is nonzero.
- Format:

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

The *until* command

- until works like the while command, except it execute the loop if the exit status is nonzero (i.e., the command failed).
- Format:

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

for loop

- for loops allow the repetition of a command for a specific set of values
- Syntax:

  for var in value1 value2 ...
  do
  command_set
  $done$

  - command_set is executed with each value of var (value1, value2, ...) in sequence

for loop

```
for file in *.c
do
        cp $file ${file}.bak
        echo $file copied to
$file.bak
done
```

for loop

```
for a in 1 2 3 4 5 6 7 8 9 10
do
        echo $a
done
```

```
for a in {0..9}
do
        echo $a
done
```

```
for a in “one” “two” “three” “four”
do
        echo $a
done
```

```
for a in {a..p}
do
        echo $a
done
```

for loop

```
for a in {5..0}
do
        echo $a
done
```

```
#for loop c Style
for (( a=0 ; a<=10; a++))
do
        echo $a
done
```

```
#for loop with range increment by 5
for a in {0..100..5}
do
        echo $a
done
```

```
#infinite for loop
for (( ;  ; ))
do
        commands
echo ctrl+c to exit
done
```

Break and continue statement

- When control encounter $break$ statement, it will come out from the current loop.

- When control encounter $continue$ statement, it will continue with the loop without going to next line.

Prime Numbers

```
#!/bin/bash
# this script will check whether the inputted number is prime or not
echo enter a natural number
read num
flag=0
count=2
while [ $count -lt $num ]
do
if [ $(echo "$num % $count"|bc) -eq 0 ]
then
flag=1
break
fi
echo $count
count=$(echo "$count + 1"|bc)
done
if [ $flag -eq 1 ]
then
echo non prime
else
echo Prime Number
fi
```

```
#!/bin/bash
# this script will check whether the inputted number is prime or not
echo enter a natural number
read num
flag=0
count=2
while [ $count -lt $num ]
do
if [ $(echo "$num % $count"|bc) -eq 0 ]
then
flag=1
break
fi
echo $count
count=$(echo "$count + 1"|bc)
done
if [ $flag -eq 1 ]
then
echo non prime
else
echo Prime Number
fi
```

```
#!/bin/bash
# print all permutation of 1, 2 and 3
for (( i=1; i<=3; i++ ))
do
for (( j=1; j<=3; j++ ))
do
for (( k=1; k<=3; k++ ))
do
if [ $i -eq $j ] || [ $j -eq $k ] || [ $i -eq $k ]
then
continue
fi
echo $i $j $k
done
done
done
```

```
#!/bin/bash
# Print fibonacci series upto 10 terms
t1=0
t2=1
echo $t1
echo $t2
for i in {3..10}
do
t3=$(echo "$t1 + $t2"|bc)
echo $t3
t1=$t2
t2=$t3
done
```

```
#!/bin/bash
# Print fibonacci series upto 10 terms
t1=0
t2=1
echo $t1
echo $t2
for i in {3..10}
do
t3=$(echo "$t1 + $t2"|bc)
echo $t3
t1=$t2
t2=$t3
done
```

Basic Operation on String

- Define a string variable
- $x = shell\ y = \text{"shell script"}\ cmd = \$(ls)$
- Define a string variable value
- $echo\ \$x \quad echo\ \${x}$
- Finding the length of a string
- $xlength = \${\#x}$
- Concatenation of two string
- $xy = \$x\$y \quad xy = \${x}\${y}$

Basic Operation on String

- Convert string into uppercase/lowercase
- $xU = \${x\^\^}\ xL = \${x,,}$
- Replacing the part of the string using variables
- $xrep = \${x/rep/newstring}$
- Slicing the string/substring
- $\${var:pos:length}$

Array in shell script

- Creating an array
- $arr[index0] = value1$
- $arr[index1] = value2$
- …
- $Arr = (value1\ value2\ value3\ value4)$
- $Arr = ([index1] = value1\ [index2] = value2\ [index3] = value3\ [index4] = value4)$

Array in shell script

- Access array elements
- $\$\{arr[index]\}$
- Access entire array
- $\$\{arr[@]\}$ or $\$\{arr[*]\}$