Shell Scripting

Introduction

What is a Shell?

A shell is a command-line interpreter that translates commands entered by the user into a language understood by the kernel.

What is a Variable?

A variable is a character string to which we assign a value. The value could be a number, text, filename, device, or any other type of data. Simply put, it is a pointer or reference to actual data.

Shell Scripting Basics

What is Shell Scripting?

Shell scripting is writing a series of command for the shell to execute. It can automate tasks, perform complex operations, and manage system functions.

Why Use Shell Scripts?

- **Automation**: Automate repetitive tasks.
- **Customization**: Customize your environment.
- **Batch Processing**: Execute a series of commands as a single job.
- **Simplification**: Simplify complex command sequences.

Steps to Write a Shell Script

1. **Open a text editor**: Use any text editor such as vi editor.
2. **Create a new file**: Save it with a .sh extension (e.g., myscript.sh).
3. **Start with the shebang**: The first line should be **#!/bin/bash** to specify the script should be run in the bash shell.
4. **Write your commands**: Add the commands you want to automate.
5. **Make the script executable**: Use the command chmod +x myscript.sh.
6. **Run the script**: Execute it by typing ./myscript.sh.

Shebang (#!): The shebang line at the beginning of a script specifies the interpreter to be used to execute the script.

Examples:

#!/bin/bash
#!/bin/sh
#!/bin/dash

Common Commands

- ls: List directory contents.
- cd: Change directory.
- pwd: Print working directory.
- cp: Copy files or directories.
- mv: Move/rename files or directories.
- rm: Remove files or directories.

Basic Shell Script

Sample Script

```
#!/bin/bash
# AUTHOR: Madhu Kiran
# DATE: 25 JULY 24
# VERSION: V1.0
# DESCRIPTION: XYZ

cd /opt
echo "ENTER THE NAME"
read name
mkdir $name
cd $name
touch jenkins ansible docker terraform linux
mkdir k8s terraform
cd k8s
touch argocd
```

Command Examples

Reading Input

Use the read command to get input from the user.

Example:

```
#!/bin/bash
echo "Enter your name:"
read name
echo "Hello, $name!"
```

Working with Files

```
#!/bin/bash
cd /opt
echo "Creating directory structure and files"
mkdir project
cd project
touch file1.txt file2.txt
mkdir dir1 dir2
```

Text Processing

Using cut and awk for text processing:

```
#!/bin/bash
# Using cut
cat /etc/passwd | cut -d ":" -f 1,6

# Using awk
cat /etc/passwd | awk -F":" '{print $1 "=" $6}'
cat /etc/passwd | awk -F":" '{print $1}'
```

Positional Parameters

Special variables referring to positional parameters:

- `$0`: This represents the name of the script itself.
- `$1`, `$2`, ..., `$9`: These represent the first, second, ..., and ninth arguments passed to the script, respectively. You can use numbers beyond `$9` by enclosing them in curly braces, like `${10}`, `${11}`, etc.
- `$#`: This gives the total number of arguments passed to the script.
- `$@`: This represents all the arguments passed to the script as separate words. It treats each argument as an individual word, which is particularly useful when you need to iterate over arguments.

- `$*`: This represents all the arguments passed to the script as a single word. It concatenates all the arguments into a single string.

- `$?`: Exit status of the previous command

Example:

```
#!/bin/bash
echo "The name of this script is: $0"
echo "The first argument is: $1"
echo "The second argument is: $2"
```

Examples of Using Positional Parameters

Example 1: Basic Script Using Positional Parameters

Here's a simple shell script that demonstrates the use of positional parameters:

```
#!/bin/bash

# Script name
echo "The name of this script is: $0"

# First and second arguments
echo "The first argument is: $1"
echo "The second argument is: $2"

# Total number of arguments
echo "Total number of arguments: $#"

# All arguments
echo "All arguments: $@"
```

Running the Script

Save the script as example.sh, and run it as follows:

```
./example.sh arg1 arg2
```

Output:

```
The name of this script is: ./example.sh

The first argument is: arg1

The second argument is: arg2

Total number of arguments: 2

All arguments: arg1 arg2
```

Example 2: Using Positional Parameters in a Loop

You can use $@ to loop through all the arguments passed to a script:

```
#!/bin/bash

echo "All arguments passed to the script are:"

# Loop through all arguments
for arg in "$@"; do
    echo "$arg"
done
```

Running the Script:

```
./example.sh one two three
```

Output:

```
All arguments passed to the script are:
one
two
three
```

Example 3: Using $* vs $@

The difference between $* and $@ can be illustrated with a simple example:

```
#!/bin/bash

echo "Using *: "
for arg in "$*"; do
    echo "$arg"
done

echo "Using @: "
for arg in "$@"; do
    echo "$arg"
done
```

Running the Script

```
./example.sh "first arg" "second arg" "third arg"
```

Output:

```
Using *:
first arg second arg third arg
Using @:
first arg
second arg
```

```
third arg
```

Debugging and Error Handling

Debugging

set -x # Turn on debugging
set +x # Turn off debugging

Error Handling

set -e # Exit on error

```
#!/bin/bash
set -e
```

Common Scripts

Jenkins Installation Script

```
sudo yum update –y
sudo wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-stable/jenkins.repo
sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io-2023.key
sudo yum upgrade
sudo dnf install java-17-amazon-corretto -y
sudo yum install jenkins -y
sudo systemctl enable jenkins
sudo systemctl start jenkins
```

Tomcat Installation Script

```
cd /opt
sudo wget https://dlcdn.apache.org/tomcat/tomcat-9/v9.0.91/bin/apache-tomcat-9.0.91.tar.gz
sudo tar -xvf /opt/apache-tomcat-9.0.91.tar.gz
cd /opt/apache-tomcat-9.0.91/webapps/manager/META-INF
sudo sed -i 's/"127\\.\\d+\\.\\d+\\.\\d+|::1|0:0:0:0:0:0:0:1"/".*"/g' context.xml
cd /opt/apache-tomcat-9.0.91/conf
sudo mv tomcat-users.xml bkp_tomcat-users.xml_23Apr24
sudo touch tomcat-users.xml
sudo echo '<?xml version="1.0" encoding="utf-8"?>
<tomcat-users>
<role rolename="manager-gui"/>
<user username="tomcat" password="tomcat" roles="manager-gui, manager-script, manager-
status"/>
</tomcat-users>' > tomcat-users.xml
```

```
sudo sed -i 's/Connector port="8080"/Connector port="8091"/g' server.xml
sudo /opt/apache-tomcat-9.0.91/bin/startup.sh
```

Reading User Input

```
#!/bin/bash
echo "Enter 1st number"
read x
echo "Enter 2nd number"
read y
sum=$((x + y))
echo $sum
```

Variables

Types of Variables

- Local Variables
- Environment Variables
- Shell Variables

Example of Variables

```
#!/bin/bash

name="Madhu kiran"

echo $name

unset name

echo $name  # Output: (nothing, the variable is unset)
```

Basic Operators

Arithmetic Operators

```
a=10
b=5

# Addition
sum=$((a + b))
echo $sum  # Output: 15
```

```
# Subtraction
difference=$((a - b))
echo $difference  # Output: 5



# Multiplication
product=$((a * b))
echo $product  # Output: 50


# Division
quotient=$((a / b))
echo $quotient  # Output: 2


# Modulus
remainder=$((a % b))
echo $remainder  # Output: 1
```

Relational Operators

```
a=10
b=10
if [ $a -eq $b ]; then
  echo "a is equal to b"
fi



if [ $a -ne $b ]; then
  echo "a is not equal to b"
fi



if [ $a -gt $b ]; then
  echo "a is greater than b"
fi



if [ $a -ge $b ]; then
  echo "a is greater than or equal to b"
fi
```

```
if [ $a -lt $b ]; then
  echo "a is less than b"
fi



if [ $a -le $b ]; then
  echo "a is less than or equal to b"
fi
```

String Comparison Operators

```
str1="hello"
str2="hello"
if [ "$str1" = "$str2" ]; then
  echo "str1 is equal to str2"
fi

if [ "$str1" != "$str2" ]; then
  echo "str1 is not equal to str2"
fi


str=""
if [ -z "$str" ]; then
  echo "str is empty"
fi


str="hello"
if [ -n "$str" ]; then
  echo "str is not empty"
fi
```

File Comparison Operators

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


if [ -f "file.txt" ]; then
  echo "file.txt is a regular file"
```

```
fi
if [ -d "directory" ]; then
  echo "directory is a directory"
fi


if [ -r "file.txt" ]; then
  echo "file.txt is readable"
fi
```

Boolean Operators

```
#!/bin/bash
a=10
b=20
c=30

if [ $a -lt $b ] && [ $b -lt $c ] || [ $a -eq 10 ]; then
  echo "a is less than b and b is less than c, or a is equal to 10"
fi
```

Local Variables

Local variables are defined within a specific function or script and are temporary, meaning they only exist during the execution of that particular function or script.

- **Definition**: Declared inside a function or a script.
- **Scope**: Only accessible within the function or script where they are defined.
- **Example**:

```
function myFunction() {

    local myVar="Hello"

    echo $myVar

}

myFunction  # Output: Hello

echo $myVar  # Output: (no output, variable is not accessible here)
```

Environment Variables

Environment variables are set using the export command, making them available system-wide and accessible by any child process.

- **Definition**: Set using the export command.
- **Scope**: Available to the current shell session and any child processes.
- **Example**:

```
export MY_ENV_VAR="World"

echo $MY_ENV_VAR  # Output: World
```

Shell Variables

Shell variables are defined in the current shell session. If not exported, they are not accessible to child processes.

- **Definition**: Declared without using export.
- **Scope**: Only available in the current shell session.
- **Example**:

```
MY_SHELL_VAR="Hello Shell"

echo $MY_SHELL_VAR  # Output: Hello Shell
```

Reading Variables

To read a variable, you can use the read command. This is commonly used to take user input and store it in a variable.

- **Command**:

  read VARIABLE_NAME

- **Example**:

```
echo "Enter your name: "

read name

echo "Hello, $name!"
```

Unsetting Variables

To remove a variable, use the unset command. This can be used to unset local, shell, or environment variables.

- **Command**:

  unset VARIABLE_NAME

- **Example**:

```
myVar="Goodbye"

unset myVar

echo $myVar  # Output: (no output, variable is unset)
```

Documentation Example

```
#!/bin/bash
# Reading user input
echo "Enter your favorite color: "
read favorite_color
echo "Your favorite color is $favorite_color"

# Unsetting the variable
unset favorite_color
echo "After unsetting, favorite_color is: $favorite_color"
```

Script to find length of s string :

```
#!/bin/bash

str="Madhukiran"

length=${#str}
echo "The length of the string is: $length"
```

Conditional Statements

Example1:

```
#!/bin/bash

number=7
```

```
if [ "$number" -gt 10 ]; then
   echo "The number is greater than 10."

elif [ "$number" -eq 10 ]; then
   echo "The number is exactly 10."

else
   echo "The number is less than 10."
Fi
```

Example2:

```
#!/bin/bash

age=20

if [ "$age" -ge 18 ]; then
   echo "You are a Major."
else
   echo "You are a minor."
fi
```

Loops

Shell loops are used to execute a set of commands repeatedly based on a condition or over a list of items. There are several types of loops in shell scripting:

1. **For Loop**
2. **While Loop**
3. **Until Loop**
4. **Nested Loop**
5. **Loop Control** (Break, Continue)
6. **Infinite Loop**

1. For Loop

The for loop iterates over a list of items, executing commands for each item in the list.

Syntax

```
for var in list
do
  statements
done
```

Example

```
#!/bin/bash

for i in 1 2 3 4 5 6 7 8 9 10
do
```

```
 echo $i
done
```

Example: Iterate from 1 to 100

```
#!/bin/bash

for var in {1..100}
do
 echo "$var"
done
```

2. While Loop

The while loop executes a set of commands as long as the condition is true. Once the condition fails, it exits the loop.

Syntax

```
while [ condition ]
do
 statements
done
```

Example

```
#!/bin/bash

i=0
while [ "$i" -le 100 ]
do
 echo "$i"
 i=$((i+1))
done
```

Example: Printing in Reverse Order

```
#!/bin/bash

i=100
while [ "$i" -ge 0 ]
do
 echo "$i"
 i=$((i-1))
done
```

3. Until Loop

The until loop executes a set of commands until the condition becomes true.

Syntax

```
until [ condition ]
do
 statements
done
```

Example

```
#!/bin/bash
```

```
i=0
until [ "$i" -gt 100 ]
do
  echo "$i"
  i=$((i+1))
done
```

4. **Infinite Loop**

An infinite loop runs indefinitely. It can be created using the while or until loop.

Example: Using Until Loop

```
#!/bin/bash

i=1
until [ "$i" -lt 0 ]
do
  echo $i
  i=$((i+1))
done
```

5. **Loop Control**

- **Break Statement**

The break statement is used to exit from a loop.

Example

```
#!/bin/bash

for var in {1..100}
do
  if [ "$var" -eq 5 ]
  then
    break
  fi
  echo $var
done
```

- **Continue Statement**

The continue statement is used to skip the current iteration of the loop and move to the next iteration.

Example

```
#!/bin/bash

for var in {1..100}
do
  if [ "$var" -eq 5 ]
  then
    continue
```

```
 fi
 echo $var
done
```

6. Nested Loop

Nested loops in shell scripting are loops within loops. You can use nested loops to perform more complex tasks that involve multiple levels of iteration.

```
#!/bin/bash

a=0
while [ "$a" -lt 5 ]

do
 b="$a"
  while [ "$b" -ge 0 ]
    do
      echo -n "$b "
      b=$((b-1))

    done
  echo
  a=$((a+1))

done
```

Output:

0
1 0
2 1 0
3 2 1 0
4 3 2 1 0

Functions

- Functions in shell scripting allow you to group a set of commands into a single, reusable unit.
- Break down total functionality of script into logical sub-sections
- Encapsulate and reuse code
- We can call functions when ever needed
- Make your scripts more organized, modular, and easier to maintain.

Defining Functions

In shell scripts, a function is defined using the following syntax:

```
function_name() {
  commands
}
```

Or alternatively:

```
function function_name {
  commands
}
```

Using Functions

Once defined, you can call a function by simply using its name:

```
function_name
```

Example

Here's a simple example of a shell script that defines and uses a function:

```
#!/bin/bash
# Define a function
greet() {
  echo "Hello, $1!"
}
# Call the function
greet "World"
greet "Madhu"
```

Functions with Return Values

Functions can return values using the return command, which returns an exit status (an integer between 0 and 255). You can capture this in a variable using $?.

```
#!/bin/bash
# Define a function with return value
```

```
calculate_sum() {
  local sum=$(( $1 + $2 ))
  return $sum
}
# Call the function
calculate_sum 5 10
# Capture the return value
result=$?
echo "The sum is $result"
```

Passing Arguments to Functions

You can pass arguments to functions just like you would with a script. Inside the function, you can access the arguments using $1, $2, etc.

```
#!/bin/bash
# Define a function that uses arguments
multiply() {
  local result=$(( $1 * $2 ))
  echo $result
}
# Call the function with arguments
product=$(multiply 4 5)
echo "The product is $product"
```

REAL TIME AUTOMATION

1. Generating HEALTH CHECK Reports:

```
#!/bin/bash

generate_report() {
    local report_file="/opt/system_health_$(date +%F).txt"
    echo "System Health Report - $(date)" > "$report_file"
    echo "----------------------------" >> "$report_file"
    echo "Disk Usage:" >> "$report_file"
    df -h >> "$report_file"
    echo >> "$report_file"
    echo "Memory Usage:" >> "$report_file"
    free -h >> "$report_file"
    echo >> "$report_file"
    echo "UPTIME & CPU Load:" >> "$report_file"
    uptime >> "$report_file"
    echo "Report generated: $report_file"
}

# Generate and save the report
generate_report
```

2. Automating Backup Operations

```
#!/bin/bash

backup_directory() {
    local dir=$1
    local backup_name=$(basename "$dir")_$(date +%F).tar.gz
    tar -cvf "/opt/backupdir/$backup_name" "$dir"
    echo "Backup of $dir completed: $backup_name"
}


backup_directory "/opt/functions"
```

3. Deploying Applications

```
#!/bin/bash

deploy_app() {
    local app_name=$1
    local env=$2
    echo "Deploying $app_name to $env..."

    # Simulate deployment steps

    echo "Pulling latest code for $app_name..."
    echo "Building $app_name..."
    echo "Deploying $app_name to $env environment..."
    echo "SUCCESS"
    echo
}

deploy_app "fb_app" "lab"
```

4. Monitoring Disk Space

```
#!/bin/bash

check_disk_usage() {
    local threshold=$1
    local usage=$(df / | grep / | awk -F" " '{print $5}' | sed 's/%/ /g'
)

    if [ "$usage" -gt "$threshold" ]; then
        echo "Disk usage is above ${threshold}%: ${usage}% used."

        # Add the code here to send an alert

    else
        echo "Disk usage is under control: ${usage}% used."
    fi
}
```

```
# Check disk usage with a threshold of 80%
check_disk_usage 80
```

5. Rolling Restart of Services

```
#!/bin/bash

restart_service() {
    local server=$1
    local service=$2

    echo "Restarting $service on $server..."

    ssh "$server" "systemctl restart $service"

}

# List of servers
servers=("fbserver1" "fbserver2" "fbserver3")


for server in "${servers[@]}"; do
    restart_service "$server" "facebook_service"
    sleep 10
done
```

1. stdout (Standard Output)

The default destination where a command sends its output data.By default, stdout is directed to the terminal or console.

```
 echo "Hello, World!"

 echo "Hello, World!" > output.txt
```

2. stderr (Standard Error)

It is used for outputting error messages or diagnostics. Unlike stdout, which is for normal output, stderr is specifically for error messages.

```
    ls /nonexistent_directory

    ls /nonexistent_directory 2> error.log
```

3. stdin (Standard Input)

stdin stands for Standard Input. It is the default source from which a command reads its input data. By default, stdin is taken from the keyboard.

&> /dev/null

&> /dev/null: This part redirects both the standard output (stdout) and standard error (stderr) of the ping command to /dev/null. /dev/null is a special file that discards any data written to it, effectively silencing all output and errors.

Example:

vim checkhosts.sh

```
#!/bin/bash

for i in $@

do
#       ping -c 1 $i &> /dev/null
        ping -c 1 $i &> /dev/null


if [ "$?" -ne 0 ]; then

    #echo "date ping failed $i host in down" | mail -s "$i host is down" gorekarmadhu@gmail.com

    echo "Date: $(date) - ping is  failed for $i -  host in down"

fi
done
```

TRAP:

The trap command in shell scripting is used to specify commands that should be executed when the shell receives certain signals or when certain events occur.

Common Signals:

SIGINT (2): Interrupt from keyboard (usually Ctrl+C).

SIGTERM (15): Termination request.

SIGHUP (1): Hangup detected on controlling terminal or death of parent process.

SIGQUIT (3): Quit from keyboard.

SIGKILL (9): Kill signal (cannot be trapped).

Example:

```
#!/bin/bash

cleanup() {
    echo "Cleaning up..."
    rm -f /tmp/tempfile
}

# Trap SIGINT (Ctrl+C) and call the cleanup function
trap 'cleanup' SIGINT


touch /tmp/tempfile


echo "Script is running. Press Ctrl+C to interrupt."
while true; do
    sleep 1
done
```