Bash Scripting

A Bash script is a file containing a sequence of commands for the Bash shell (Bourne Again Shell). Bash is the default shell on most Unix-like operating systems (e.g., Linux, macOS). Bash scripts automate tasks, simplify system administration, and handle repetitive actions.

Key Concepts:

1. Automation: Instead of typing commands one by one, you write them in a script to execute multiple commands at once.

2. Text File: A Bash script is simply a plain text file, typically with a `.sh` extension (optional but recommended).

3. Shebang (`#!`): This line (`#!/bin/bash`) tells the system to use the Bash interpreter to execute the script.

4. Permissions: To run a script, it must be executable. Use `chmod +x script.sh` to make a script executable.

5. Variables: Variables store data like strings and numbers for reuse in scripts.

6. Control Flow: Bash supports if-else, loops (`for`, `while`), and case statements to manage logic.

7. Execution: After writing and saving the script, execute it with `./script.sh` (or `bash script.sh`).

Example Script:

```
#!/bin/bash

# A simple Bash script
echo "Hello, World!" # Print a message

# Define a variable
name="John"
echo "Hello, $name!" # Use variable
```

```
# If-Else statement
if [ "$name" == "John" ]; then
echo "Welcome, John!"
else
echo "You are not John!"
fi
```

Creating and Executing a Bash Script

1. Checking your Shell:

```
echo $SHELL # Displays which shell you're using
```

2. Creating a Bash Script:

```
nano my_script.sh # Create and edit a script
```

3. Making the Script Executable:

```
chmod +x my_script.sh # Give execute permissions
```

4. Running the Script:

```
./my_script.sh # Execute the script
```

Variables in Bash

Variables store data for use in the script. Note: There should be no spaces around the equal sign when assigning values.

Declaring and Using Variables:

```
name="John" # Declare a variable
echo $name # Access the variable
```

Example:

```
my_name="Rakesh"
age=26

# Using variables
echo "My name is $my_name"
echo "My age is $age"
```

System Variables:

Bash has predefined environment variables like $USER, which holds the current user’s name.

```
echo "The system user is $USER"
```

Difference Between Single and Double Quotes:

Double quotes: Expands variables and command outputs.
Single quotes: Treats everything as literal text.

```
echo "My name is $my_name" # Expands $my_name
echo 'My name is $my_name' # Outputs "$my_name" literally
```

Mathematical Operations

Use the expr command to perform calculations in Bash. Ensure spaces around operators.

```
expr 10 + 5 # Addition
expr 15 - 5 # Subtraction
expr 5 \* 3 # Multiplication (escape * with backslash)
expr 20 / 4 # Division
```

Example:

```
num1=15
num2=5
echo "The sum of $num1 and $num2 is $(expr $num1 + $num2)"
```

Control Flow (If-Else Statements)

Bash uses if, elif, and else for conditional checks.

```
if [ $num -eq 10 ]; then
echo "Number is 10"
else
echo "Number is not 10"
fi
```

Comparisons

- `eq`: Equal to
- `ne`: Not equal to
- `gt`: Greater than
- `lt`: Less than

```
# Check if a number is greater than 100
if [ $num -gt 100 ]; then
echo "Greater than 100"
else
echo "Less than or equal to 100"
fi
```

File and Directory Checks:

```
# Check if a file exists
if [ -f filename ]; then
echo "File exists"
else
echo "File does not exist"
fi
```

Loops in Bash

Loops allow repetitive execution of code.

While Loop

```
# Check if a file exists
if [ -f filename ]; then
echo "File exists"
else
echo "File does not exist"
fi
```

For Loop:

```
for i in {1..5}; do
echo "Loop number $i"
done
```

File Loop:

```
for file in *.txt; do
echo "Processing $file"
done
```

Exit Codes

Exit codes (status codes) indicate whether a command was successful (0) or failed (1).

Check Exit Code:

```
if [ $? -eq 0 ]; then
echo "Command was successful"
else
echo "Command failed"
fi
```

Error Handling with Exit Codes

You can also store the exit status of commands to handle errors more effectively.

```
package="htop"
sudo apt install -y $package
if [ $? -ne 0 ]; then
echo "Installation failed"
else
echo "Package installed successfully"
fi
```

Redirecting Output and Errors

```
>: Redirect standard output (overwrite).
>>: Append standard output.
2>: Redirect standard error.
&>: Redirect both output and error.
```

```
# Redirect output to file
echo "Hello, World!" > output.txt

# Redirect errors to a file
find /etc -type f 2> error.log

# Redirect both output and error to the same file
find /etc -type f &> output_and_error.log
```

Functions in Bash

Functions help modularize code, making it reusable and easier to maintain.

```
# Define a function
my_function() {
echo "This is a function"
}

# Call the function
my_function
```

Example with Error Handling:

```
check_error() {
if [ $? -ne 0 ]; then
echo "An error occurred!"
fi
}

# Usage in script
sudo apt update
check_error
```

Data Streams

In Bash scripting, a data stream refers to the flow of data from one point to another. Data streams are commonly used for communication between processes

```
#!/bin/bash

# Explanation:
# This script demonstrates stdin, stdout, stderr, and data
redirection in Bash.

# 1. Simulate reading from stdin (Standard Input):
echo "Please enter your name:"
read name # Reading user input

# 2. Simulate stdout (Standard Output) - Writing the output to a
file:
echo "Hello, $name!" > output.txt # The greeting is written to
output.txt

# 3. Simulate stderr (Standard Error) - Writing an error message:
echo "This is an error message." >&2 # The error message is sent
to stderr

# 4. Redirecting both stdout and stderr to the same file:
ls /nonexistentdirectory > combined_output.log 2>&1 # This
command fails and sends both stdout and stderr to
combined_output.log

# 5. Simulating piping output - Using pipe to send output from
one command to another:
echo "This is a test" | grep "test" > pipe_output.txt # The
string is piped to grep, which filters it and writes the result
to pipe_output.txt

# 6. Showing the contents of the files created:
echo "Contents of output.txt:"
cat output.txt # Displaying the content of output.txt

echo "Contents of combined_output.log:"
cat combined_output.log # Displaying the content of
combined_output.log

echo "Contents of pipe_output.txt:"
cat pipe_output.txt # Displaying the content of pipe_output.txt
```

stdin: The input stream from which the script receives data (user input, file content).

stdout: The output stream used by the script to print results (to the terminal or to a file).

stderr: The error stream used by the script to print error messages.

Redirection: The process of sending output to files or other destinations.

Pipe: A way to send the output of one command to the input of another.

Updating Scripts for Multiple Distributions

You can write scripts that check the Linux distribution and perform different actions based on the result.

```
#!/bin/bash

release_file=/etc/os-release

if grep -q "Arch" $release_file; then
sudo pacman -Syu
fi

if grep -q "Ubuntu" $release_file || grep -q "Debian"
$release_file; then
sudo apt update && sudo apt dist-upgrade
fi
```

Adding Scripts to PATH

To make your script accessible from anywhere:

```
# Move it to /usr/local/bin:
sudo mv script.sh /usr/local/bin/

# Change its permissions:
sudo chmod +x /usr/local/bin/script.sh

# Add /usr/local/bin to your PATH:
export PATH=$PATH:/usr/local/bin
```

Passing Arguments in Bash:

You can write scripts that check the Linux distribution and perform different actions based on the result.

You can pass arguments to a Bash script when running it from the command line. These arguments can be accessed using special variables:

$0: Script name.
$1, $2, $3, etc.: First, second, third arguments.
$#: Total number of arguments.
$@: All arguments as a list.
$*: All arguments as a single string.

```bash
#!/bin/bash

# Accessing passed arguments
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "Total arguments: $#"
echo "All arguments: $@"

# Conditional check
if [ $# -lt 2 ]; then
echo "You need at least 2 arguments!"
else
echo "Arguments passed correctly!"
fi
```

Save the script to a file (e.g., args_example.sh).

Give execute permission: `chmod +x args_example.sh`

Run the script with arguments: `./args_example.sh arg1 arg2`