Basics of Linux

In Linux, process automation relies heavily on shell scripting. This involves creating a file containing a series of commands that can be executed together. In this article, we'll start with the basics of bash scripting which includes variables, commands, inputs/ outputs, and debugging. We'll also see examples of each along the way.

Introduction

A bash script is a file containing a sequence of commands that are executed by the bash program line by line. It allows you to perform a series of actions, such as navigating to a specific directory, creating a folder, and launching a process using the command line.

By saving these commands in a script, you can repeat the same sequence of steps multiple times and execute them by running the script.

Advantages of Bash scripting

Bash scripting is a powerful and versatile tool for automating system administration tasks, managing system resources, and performing other routine tasks in Unix/Linux systems. Some advantages of shell scripting are:

Automation: Shell scripts allow you to automate repetitive tasks and processes, saving time and reducing the risk of errors that can occur with manual execution.

Portability: Shell scripts can be run on various platforms and operating systems, including Unix, Linux, macOS, and even Windows through the use of emulators or virtual machines.

Flexibility: Shell scripts are highly customizable and can be easily modified to suit specific requirements. They can also be combined with other programming languages or utilities to create more powerful scripts.

Accessibility: Shell scripts are easy to write and don't require any special tools or software. They can be edited using any text editor (vim, nano etc.), and most operating systems have a built-in shell interpreter.

Integration: Shell scripts can be integrated with other tools and applications, such as databases, web servers, and cloud services, allowing for more complex automation and system management tasks.

Debugging: Shell scripts are easy to debug, and most shells have built-in debugging and error-reporting tools that can help identify and fix issues quickly.

How to Create and Execute Bash scripts

Bash scripts end with .sh. However, bash scripts can run perfectly fine without the sh extension.

Adding the Shebang

Bash scripts start with a shebang. Shebang is a combination of bash # and bang ! followed by the bash shell path. This is the first line of the script. Shebang tells the shell to execute it via bash shell. Shebang is simply an absolute path to the bash interpreter.

Below is an example of the shebang statement.

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

You can find your bash shell path (which may vary from the above) using the command:

```
which bash
```

Creating our first bash script

Our first script prompts the user to enter a path. In return, its contents will be listed.

Create a file named run_all.sh using the vim command. You can use any editor of your choice.

```
Terminal $: vim start.sh
```

Add the following commands in your file and save it:

```
#!/bin/bash
echo "Today is " `date`

echo -e "\nenter the path to directory"
read the_path
```

```
echo -e "\n you path has the following files and folders: "
ls $the_path
```

This script prints the content of a user supplied directory. Let's take a deeper look at the script line by line. I am displaying the same script again, but this time with line numbers.

```
1 #!/bin/bash
2 echo "Today is " `date`
3
4 echo -e "\nenter the path to directory"
5 read the_path
6
7 echo -e "\n you path has the following files and folders: "
8 ls $the_path
```

Line #1: The shebang (#!/bin/bash) points toward the bash shell path.

Line #2: The echo command is displaying the current date and time on the terminal. Note that the date is in backticks.

Line #4: We want the user to enter a valid path.

Line #5: The read command reads the input and stores it in the variable the_path.

line #8: The ls command takes the variable with the stored path and displays the current files and folders.

Executing the bash script

To make the script executable, assign execution rights to your user using this command:

```
chmod u+x run_all.sh-f
```

Here, chmod modifies the ownership of a file for the current user :u.

+x adds the execution rights to the current user. This means that the user who is the owner can now run the script.

run_all.sh is the file we wish to run.

You can run the script using any of the mentioned methods:

sh run_all.sh

bash run_all.sh

./run_all.sh

Comments in Bash

Comments are very helpful in documenting the code, and it is a good practice to add them to help others understand the code.

These are examples of comments:

```
# This is an example comment

# Both of these lines will be ignored by the interpreter
```

Variables and data types in Bash

Variables let you store data. You can use variables to read, access, and manipulate data throughout your script. There are no data types in Bash. In Bash, a variable is capable of storing numeric values, individual characters, or strings of characters.

In Bash, you can use and set the variable values in the following ways:

1. Assign the value directly:

   ```
   country=Pakistan
   ```

2. Assign the value based on the output obtained from a program or command, using command substitution. Note that $ is required to access an existing variable's value.

   ```
   same_country=$country
   ```

This assigns the value of country to the new variable same_country. To access the variable value, append $ to the variable name.

```
Terminal $: country=Pakistan

Terminal $: echo $country

Pakistan
```

```
Terminal $: new_country=$country
Terminal $: echo $new_country
Pakistan
```

Assigning and printing variable values

Variable naming conventions

In Bash scripting, the following are the variable naming conventions:

Variable names should start with a letter or an underscore (_).

Variable names can contain letters, numbers, and underscores (_).

Variable names are case-sensitive.

Variable names should not contain spaces or special characters.

Use descriptive names that reflect the purpose of the variable.

Avoid using reserved keywords, such as if, then, else, fi, and so on as variable names.

Here are some examples of valid variable names in Bash:

```
name
count
_var
myVar
MY_VAR
```

And here are some examples of invalid variable names:

2ndvar (variable name starts with a number)

my var (variable name contains a space)

my-var (variable name contains a hyphen)

Following these naming conventions helps make Bash scripts more readable and easier to maintain.

Input and output in Bash scripts

1. We can read the user input using the read command.

```
#!/bin/bash
echo "Today is " `date`
echo -e "\nenter the path to directory"
read the_path
echo -e "\nyour path has the following files and folders: "
ls $the_path
```

2. Command line arguments

In a bash script or function, $1 denotes the initial argument passed, $2 denotes the second argument passed, and so forth.

This script takes a name as a command-line argument and prints a personalized greeting.

```
echo "Hello, $1!"
```

We have supplied World as our argument to the script.

```
#!/bin/bash
echo "Hello, $1!"

Terminal $: ./script World
echo "Hello, World!"
```

This prints the text "Hello, World!" to the terminal.

3. Writing to a file:

```
echo "This is some text." > output.txt
```

This writes the text "This is some text." to a file named output.txt. Note that the >operator overwrites a file if it already has some content.

Appending to a file:

echo "More text." >> output.txt

This appends the text "More text." to the end of the file output.txt.

4. Redirecting output:

ls > files.txt

This lists the files in the current directory and writes the output to a file named files.txt. You can redirect output of any command to a file this way.

Conditional statements (if/else)

Expressions that produce a boolean result, either true or false, are called conditions. There are several ways to evaluate conditions, including if, if-else, if-elif-else, and nested conditionals.

Syntax:

```
if [[ condition ]];
then
        statement
elif [[ condition ]]; then
        statement
else
        do this by default
fi
```

Syntax of bash conditional statements

We can use logical operators such as AND -a and OR -o to make comparisons that have more significance.

```
if [ $a -gt 60 -a $b -lt 100 ]
```

This statement checks if both conditions are true: a is greater than 60 AND b is less than 100.

Let's see an example of a Bash script that uses if, if-else, and if-elif-else statements to determine if a user-inputted number is positive, negative, or zero:

```
#!/bin/bash


echo "Please enter a number: "
read num


if [ $num -gt 0 ]; then
  echo "$num is positive"
elif [ $num -lt 0 ]; then
  echo "$num is negative"
else
  echo "$num is zero"
fi
```

Script to determine if a number is positive, negative, or zero

The script first prompts the user to enter a number. Then, it uses an if statement to check if the number is greater than 0. If it is, the script outputs that the number is positive. If the number is not greater than 0, the script moves on to the next statement, which is an if-elif statement. Here, the script checks if the number is less than 0. If it is, the script outputs that the number is negative. Finally, if the number is neither greater than 0 nor less than 0, the script uses an else statement to output that the number is zero.

Seeing it in action

```
Terminal $: test-odd
```

Looping and Branching in Bash

While loop

While loops check for a condition and loop until the condition remains true. We need to provide a counter statement that increments the counter to control loop execution.

In the example below, ((i += 1)) is the counter statement that increments the value of i. The loop will run exactly 10 times.

```
#!/bin/bash
i=1
while [[ $i -le 10 ]] ; do
   echo "$i"
  (( i += 1 ))
done
```

For loop

The for loop, just like the while loop, allows you to execute statements a specific number of times. Each loop differs in its syntax and usage. In the example below, the loop will iterate 5 times.

```
#!/bin/bash
for i in {1..5}
do
    echo $i
done
```

How to Schedule Scripts using cron

Cron is a powerful utility for job scheduling that is available in Unix-like operating systems. By configuring cron, you can set up automated jobs to run on a daily, weekly, monthly, or specific time basis. The automation capabilities provided by cron play a crucial role in Linux system administration.

Below is the syntax to schedule crons:

Cron job example

* * * * * sh /path/to/script.sh

Here, the *s represent minute(s) hour(s) day(s) month(s) weekday(s), respectively.

Below are some examples of scheduling cron jobs.

SCHEDULE	DESCRIPTION	EXAMPLE
0 0 * * *	Run a script at midnight every day	0 0 * * * /path/to/script.sh
*/5 * * * *	Run a script every 5 minutes	*/5 * * * * /path/to/script.sh
0 6 * * 1-5	Run a script at 6 am from Monday to Friday	0 6 * * 1-5 /path/to/script.sh
0 0 1-7 * *	Run a script on the first 7 days of every month	0 0 1-7 * * /path/to/script.sh
0 12 1 * *	Run a script on the first day of every month at noon	0 12 1 * * /path/to/script.sh

Using crontab

The crontab utility is used to add and edit the cron jobs.

crontab -l lists the already scheduled scripts for a particular user.

You can add and edit the cron through crontab -e.

You can read more about corn jobs in my other article here.

How to Debug and Troubleshoot Bash Scripts

Debugging and troubleshooting are essential skills for any Bash scripter. While Bash scripts can be incredibly powerful, they can also be prone to errors and unexpected behavior. In this section, we will discuss some tips and techniques for debugging and troubleshooting Bash scripts.

Set the set -x option

One of the most useful techniques for debugging Bash scripts is to set the set -x option at the beginning of the script. This option enables debugging mode, which causes Bash to print each command that it executes to the terminal, preceded by a + sign. This can be incredibly helpful in identifying where errors are occurring in your script.

```
#!/bin/bash

set -x

# Your script goes here
```

Check the exit code

When Bash encounters an error, it sets an exit code that indicates the nature of the error. You can check the exit code of the most recent command using the $? variable. A value of 0 indicates success, while any other value indicates an error.

```
#!/bin/bash

# Your script goes here

if [ $? -ne 0 ]; then
   echo "Error occurred."
fi
```

Use echo statements

Another useful technique for debugging Bash scripts is to insert echo statements throughout your code. This can help you identify where errors are occurring and what values are being passed to variables.

```
#!/bin/bash

# Your script goes here

echo "Value of variable x is: $x"
```

# More code goes here

Use the set -e option

If you want your script to exit immediately when any command in the script fails, you can use the set -e option. This option will cause Bash to exit with an error if any command in the script fails, making it easier to identify and fix errors in your script.

```
#!/bin/bash

set -e

# Your script goes here
```

Troubleshooting crons by verifying logs

We can troubleshoot crons using the log files. Logs are maintained for all the scheduled jobs. You can check and verify in logs if a specific job ran as intended or not.

For Ubuntu/Debian, you can find cronlogs at:

```
/var/log/syslog
```

The location varies for other distributions.

A cron job log file can look like this:

```
2022-03-11 00:00:01 Task started
2022-03-11 00:00:02 Running script /path/to/script.sh
2022-03-11 00:00:03 Script completed successfully
2022-03-11 00:05:01 Task started
2022-03-11 00:05:02 Running script /path/to/script.sh
2022-03-11 00:05:03 Error: unable to connect to database
2022-03-11 00:05:03 Script exited with error code 1
2022-03-11 00:10:01 Task started
```

```
2022-03-11 00:10:02 Running script /path/to/script.sh
2022-03-11 00:10:03 Script completed successfully
```

Cron log