Scripts and Pipes

Creating Scripts

Creating a list of commands to run one after the other is as easy as writing a grocery list of things you need to buy (once you know the basic steps).
Creating scripts for commonly run sequences of commands can come in handy. You can save the commands as a text file and then run the sequence as a single step. The steps required for creating a script from a bird's eye view are as follows:

1. Create a text file containing the commands.
2. Make the top line of the file a *shebang* (explained later).
3. Save file.
4. Make the file executable using permissions (explained later).
5. Run the command `./myScript.sh`.

Below is an example of a simple script called name.sh:

```
#!/usr/bin/env bash
echo First name: $1
echo Last name: $2
```

This script takes two arguments, one for first name and one for last. These arguments are represented in the code with $1 and $2. It would be executed by running

```
./name.sh Benjamin Franklin
```

When run two lines will output, the first line being “First name: Benjamin” and the second line “Last name: Franklin”. Unless of course you swap the input for your own name in which case the names will be swapped out.

Shebang

A shebang refers to the first line in a script, when that line begins with #!. The word comes from the musical notation term for # sharp and the ! sometimes being called “bang”; combining these two becomes “sharp-bang” or shebang for short.
The shebang when used as the first line of a file specifies the program which will be used to interpret the script. The most popular one relevant to writing Linux scripts is

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

The same thing can be expressed using /usr/bin/env which increases portability by using whatever version of bash is found in the user's path.

```
#!/usr/bin/env bash
```

The shebang is not limited to bash scripts. It should also be the first line when writing scripts in other scripting languages such as python, ruby, or perl.

```
#!/usr/bin/env python
```

File Permissions

As mentioned previously, the fourth step in making an executable script is changing the permissions on the file to allow execution. The short and simple way of doing this is to run

```
chmod +x name.sh
```

This simply adds the execution permission to the file for our current user. After running the command, you'll be able to make use of it simply by running the following (assuming you're in the same directory as the file):

```
./name.sh
```

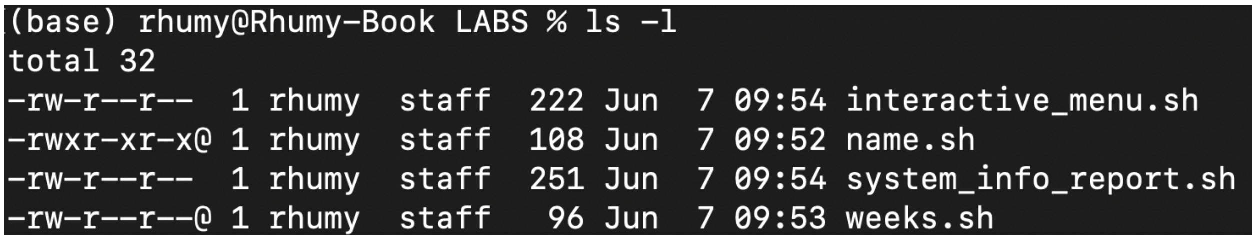
It's worth understanding the concept of permissions on Linux as it's a crucial aspect of the operating system. Every file has three different types of permissions:

- Read
- Write
- Execute

Each of these three permissions can be set separately for three groups:

- User
- Group
- Others

When using `ls -l`, you can see the set permissions for each file expressed on the left-hand side, as shown in the figure below:



```
(base) rhumy@Rhumy-Book LABS % ls -l
total 32
-rw-r--r--  1 rhumy  staff  222 Jun  7 09:54 interactive_menu.sh
-rwxr-xr-x@ 1 rhumy  staff  108 Jun  7 09:52 name.sh
-rw-r--r--  1 rhumy  staff  251 Jun  7 09:54 system_info_report.sh
-rw-r--r--@ 1 rhumy  staff   96 Jun  7 09:53 weeks.sh
```

Figure 1. Permissions for files shown in the first column when running ls -l

Note the first letter in this ten-letter sequence is used to indicate special file types. The possible values are d=directory, c=character device, s=symlink, p=named pipe, s=socket, b=block device, and D=door. We don’t have to deal with these special types, but it’s worth knowing what the first letter is.

After the first letter which indicates special file types, there are nine more letters. We can break these nine letters into three sets of three, as shown in Figure 2 – the first being file permissions for file owner, the second for user group, and the third for all other users.

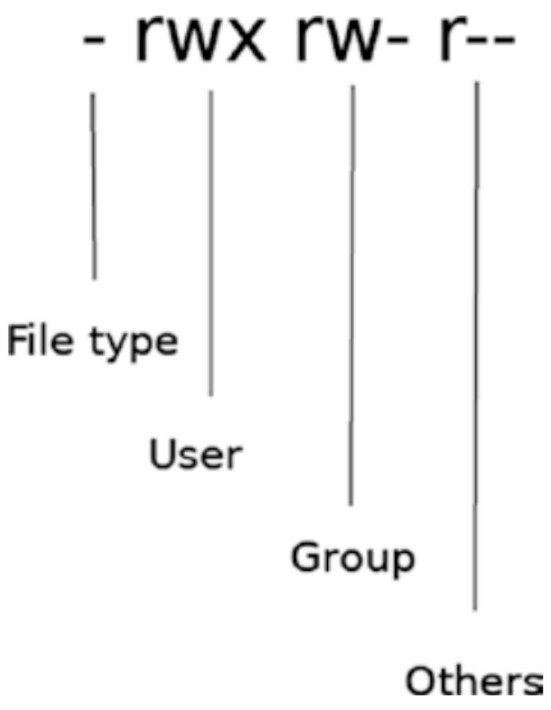


Figure 2. Components of file permissions

For the three sections, we have three different letters which if present indicate that groups has said permission:

- r = read
- w = write
- x = execute

In the example given, we have a file with all permissions for the owner, read and write for the user group, and only the ability to read for all others who can access the file.

Permission data can also be displayed as a set of three numeric symbols where a single number represents the combination of the permissions. Each of the permission types is given a numeric value:

- 4 = read
- 2 = write
- 1 = execute

For any group we add up the permissions to get a number representing allowed permissions. For example, read and write would be **`6(2+4)`**, write and execute would be **`3(1+2)`**, and no permissions would be **`0`**.

Using this notation, we would express **`rxw rw- r--`** as **`764`**. Either of these notations can be used when changing permissions for a file. For example, we can run

```
chmod 777 numbers.sh
```

This gives all permissions to all users. Or if we want to use the notation with letter, we could run the following command to take away execution permission for all groups (note the-; if we wanted to add, it would instead be +):

```
chmod -x numbers.sh
```

If we want to use number notation for a specific column (user, group, or others), we can first specify the group, for example, add execution permission back but only for the owner:

```
chmod u+x numbers.sh
```

File Types

While we've defined our script using the file type `.sh`, this is not actually required in Linux. We can just as easily have named it `name` instead of `name.sh`, and it would work just the same.

Note Some teams prefer scripts without the '.sh' extension, for example, the Google Shell Style Guide actually specifies the extension `.sh` should not be used.
Despite this several public repositories managed by Google contain shell scripts which include the `.sh`. This just goes to show even at a company which states a preference you can't be sure if scripts will include the `.sh` extension.

A useful command for detecting file type is file. To experiment with this, first change the file name of name.sh to name. Next run the following:

```
file name
```

You should get back a message saying the file type is `Bourne-Again shell script`. Next try opening the file and editing the shebang to be the one for python, as listed in the shebang section.

```
#!/usr/bin/env python
```

After saving, try running the file again. Repeat this process trying different shebangs, including python, ruby, and perl. You should get results similar to those shown in Figure 3.

Grep

`grep` stands for **Global Regular Expression Print**. It's a powerful command-line utility used to **search for text patterns** within files or input streams.

It returns lines that **match a given pattern**, making it ideal for filtering logs, files, and command outputs.

Basic Syntax

grep [options] 'pattern' filename

1. Search for a word in a file: **grep "error" logfile.txt**
2. Ignore case when searching: **grep -i "warning" system.log**

3. Show line numbers with matches: **grep -n "devops" notes.txt**

4. Recursively search in all `.txt` files in a directory: **grep -r "Linux" .**

Pipes

Pipes are one of the most common features of basic syntax. A pipe simply connects the output of one command as the input to another command.

A **pipe (`|`)** lets you **send the output of one command** as the **input to another command**, forming a chain of operations.

✅ Terminal-Ready Pipe Examples

Level 1

1. List files and count them: **`ls | wc -l`**
   (Lists all files in the current directory and counts them.)

2. Search for a word in a file: **`cat file.txt | grep "error"`**
   (Shows lines in `file.txt` that contain the word “error”.)

3. Display system users, sorted: **cut -d: -f1 /etc/passwd | sort**
   (Extracts usernames from `/etc/passwd` and sorts them.)

4. Show only the top 5 largest files: **ls -lhS | head -n 5**
(Lists files by size (human-readable), then shows the top 5.)

Level 2

5. List files, remove duplicates, count them: **ls | sort | uniq | wc -l**
(Useful if symbolic links or aliases create duplicates.)

6. Multiple pipes: Get disk usage of each subfolder and show top 3 largest: **du -h --max-depth=1 | sort -hr | head -n 3**
(Shows the top 3 largest folders in your current directory.)

7. Show the top 10 most used commands from your history:
history | awk '{print $2}' | sort | uniq -c | sort -nr | head -n 10

(Counts and displays your most frequently used commands.)

📘 **Breakdown:**

- `history` → **list of past commands**
- `awk '{print $2}'` → **extract the command name**
- `sort | uniq -c` → **count unique commands**
- `sort -nr` → **sort numerically in reverse**
- `head -n 10` → **top 10**

Project based examples
=========================

1. **Count the number of words in all .txt files**

✅ **Step 1: Create and Populate `.txt` Files**
# Create and write to file1.txt
echo "DevOps is a combination of development and operations." > file1.txt

# Create and write to file2.txt
echo "Linux commands are powerful when used with pipes and scripts." > file2.txt

# Create and write to file3.txt
echo "Automation improves reliability and reduces manual effort." > file3.txt

🔹 Each echo ... > filename.txt command:

Creates the file if it doesn’t exist

Writes the given text into it (overwriting if the file exists)

✅ Confirm the files exist:
```
ls *.txt
```

✅ Step 2: View the Contents of Each File

You can use the `cat` command to display the contents of the files in your terminal.

```
cat file1.txt file2.txt file3.txt
```

✅ Step 3: Count the Number of Words in All `.txt` Files

We'll use the `cat` and `wc -w` commands together with a **pipe (`|`)** to achieve this.

```
cat *.txt | wc -w
```

🧠 Explanation:

- `cat *.txt` → reads and combines the contents of all `.txt` files in the current directory
- `|` → pipes that combined content as input to the next command
- `wc -w` → counts the total number of **words**

Extra tasks:

1. Command that counts the number of words per file and prints out the file name and the counts: wc -w *.txt

   ### 🧠 Explanation:

   `wc` is the **word count** utility
   `-w` tells it to count **words**
   `*.txt` matches all `.txt` files in the current directory

2. To **write the word count per file** into a new file called `summary.txt: wc -w *.txt > summary.txt`

   ### 🧠 Explanation:

```
Runs wc -w on all .txt files:
Writes the output (word count + filename + total) into
summary.txt
Optional: Append Instead of Overwrite
     wc -w *.txt >> summary.txt
```

Redirects

As we saw in the last section, we can use the > character to send text into a file rather than piping into another program. This can be done with the output from any program. When using the standard redirect, you should be aware that any existing content in that file will be overwritten. Running

```
echo dog > /tmp/test
echo cat > /tmp/test
```

will result in /tmp/test only containing the text “cat”. If you want to append text to the file instead of replacing the content, you should instead use >>:

```
echo dog >> /tmp/test
echo cat >> /tmp/test
```

This will instead result in a file which contains two lines, one with “dog” and one with “cat”.

The output in a redirect by default contains both the output and any errors. We can instead redirect errors to a separate location by adding

```
echo cat > /tmp/test 2> /tmp/error
```

However, with the preceding example, no errors are being created. To generate both standard output and standard error in a single command, use ls on an existing file and a nonexistent file:

```
ls /tmp/test /tmp/nope777 > /tmp/test 2> /tmp/error
```

After running the preceding command, you should have content in both the /tmp/ test file and /tmp/error. As with a normal redirect, we can use >> to append instead of replace the text:

```
ls /tmp/test /tmp/nope777 >> /tmp/test 2>> /tmp/error
```

If you run the preceding command multiple times, you'll end up with lines in each file for each time it was run.

Redirect and Pipe at Once with tee

Redirecting output to a file and piping are both powerful tools, but what if you want to do them both at once? A popular utility called tee exists for exactly this purpose. It duplicates the input and sends it to both a file and the output as shown in Figure

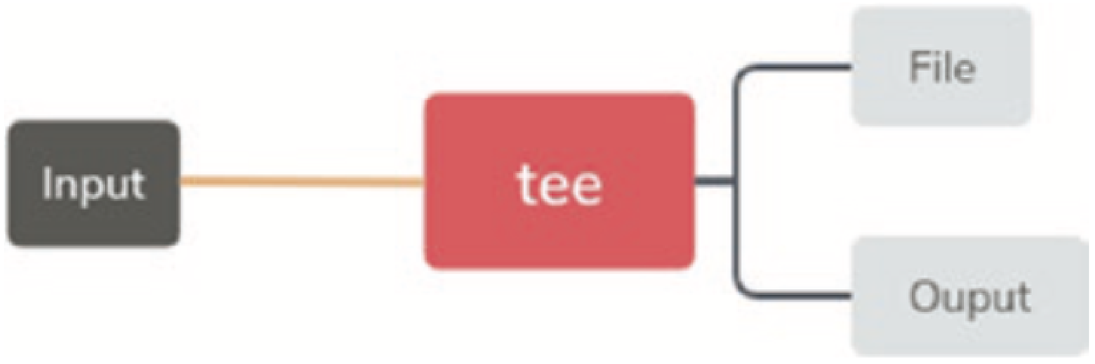


Figure 4: Diagram of output from tee command

The tee command takes output from standard output, saves it to a file of your choice, then passes that output to its own standard output. For example, say we have the following command using a redirect to write the output "hello" to a file called greeting:

```
echo hello > greeting
```

Running the preceding command, we'll end up with "hello" in our greeting file but will see nothing in our standard output. The same program modified to use tee would be

```
echo hello | tee greeting
```

With `tee`, we'll end up with "hello" in our greeting file, but we also see "`hello`" in the standard output. If you want tee to act like >> and append to a file rather than > which replaces the text, you can use the -a flag.

Another example of using tee, say we want to pass a math equation to the math utility bc for processing. We'll output the result to a file called math, but we also want to show the equation that led to the result. We could make use of tee for this using the following command:

```
echo "7 * 7" | tee math.txt | bc >> math.txt
```

This causes the file math.txt to be written twice. Once using tee and the input, and a second time via '>>'. The file math.txt should contain:
7 * 7

Conditional Expressions in Bash

As you start to combine several components of programs using && and || via the command line, you'll likely find it starts to get easier to write a dedicated script rather than manually enter a long string of commands from the command line. As you move from command line to writing a script, it'll be easier to use some of the more complicated syntax tools.
One of these tools is the if statement, which is more like a series of possible tests, each with their own specific option. For example, if we want to check if a file exists, we'd use the -e option. Create a script and add the following:

```
if [ -e /etc/passwd ]; then
    echo passwd exists
fi
```

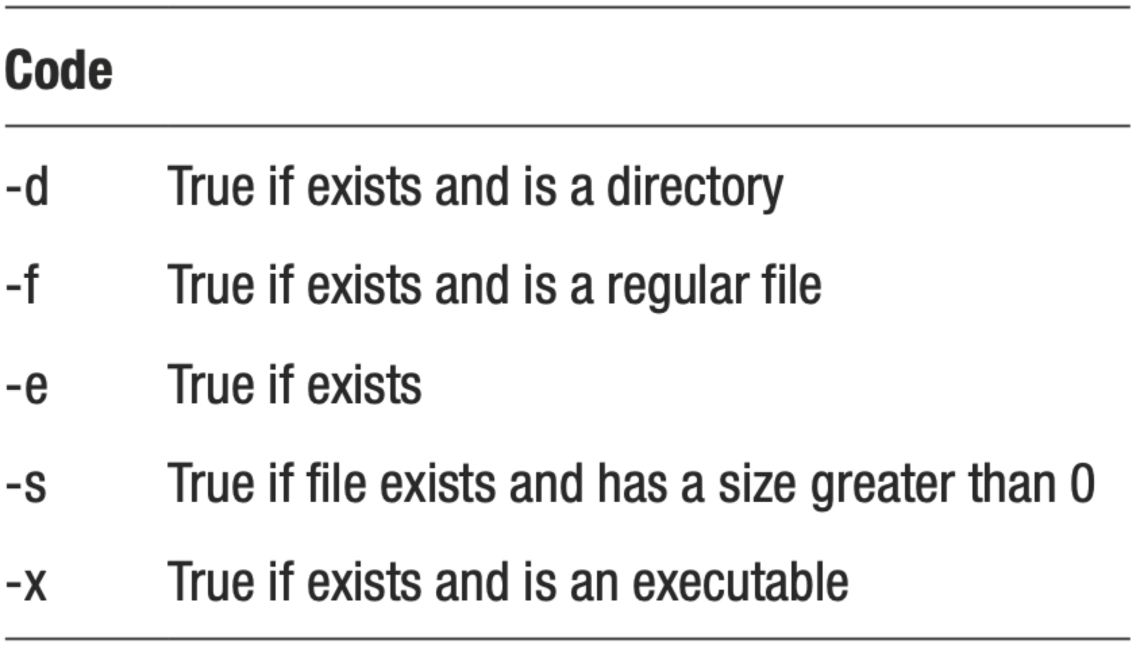
When you run the script, you should get the output "passwd exists". Try changing /etc/passwd to a file that doesn't exist. Or if you'd like to test if the file doesn't exist, you can add a ! as shown in the following:

```
if [ ! -e /etc/passwd ]; then
```

As with other languages, we can add an else to our if statement:

```
if [ -e /etc/passwd ]; then
   echo passwd exists
else
   touch /etc/passwd
fi
```

The preceding syntax works for several different possible tests that can be run by substituting the -e. The list is quite long and can be found by running man bash and scrolling down to the conditional expression section. Some of the more commonly used flags are shown in Table below:



Code	
-d	True if exists and is a directory
-f	True if exists and is a regular file
-e	True if exists
-s	True if file exists and has a size greater than 0
-x	True if exists and is an executable

Loop in Bash

Like other languages, bash also provides a way to do loops over a set of items. For example, given a set of names, we can print “`hello`” for each:

```
for name in jesse james jen
do
   echo "Hello $name"
done
```

This can be useful with the expansion we looked at in the previous section, for example:

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

It’s also possible to use an array as the data provider for a loop. Arrays are defined using bash as shown here:

```
array=( 1 39 47 )
```

Then to make use of the array, it needs to be expanded using curly braces:

```
for i in ${array[@]}
do
     echo "Hello $i"
done
```

While we’ve used integers here, you can just as easily use strings or another data type in your array.

While Loops

In some situations, you may be better off using a while loop rather than a for loop. Instead of having a set number of iterations, you might want to end the loop based on a value not related to iterations. For example, say we want to see how many times we can loop through some code in 7 seconds.

Note For the following example, you’ll need to put the code into a script file and run chmod +x to add execution permission. This is because of our use of the special variable $SECONDS. The $SECONDS variable contains the amount of seconds a terminal, or in our case, script, has been running.

```
#!/usr/bin/env bash

i=0
while [ $SECONDS -lt 7 ]; do
   i=$((i+1))
done

echo $i
```

Executing the script will return the amount of times the loop was able to run in 7 seconds. You might be surprised with how high the number is. In my case, the loop ran 927375 times.

The while loop allows us to limit the running of a code section without a specific number like the for loop. While we’ve used the example of time, you could also use some external value. For example, if a website is down, you may want to keep checking every few minutes until it is back up.