How-To Geek

Linux

9 Bash Script Examples to Get You Started on Linux

DAVE MCKAY @thegurkha
JUN 29, 2022, 11:00 AM EDT | 10 MIN READ

fatmawati achmad zaenuri/Shutterstock.com

If you're starting out with Bash scripting on Linux, getting a solid grasp of the basics will stand you in good stead. They're the foundation of deeper knowledge and higher scripting skills.

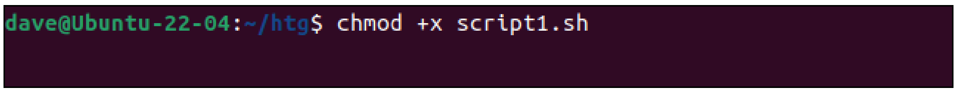
Remember, Make Your Scripts Executable

For the shell to execute a script, the script must have the executable file permission set. Without this, your script is just a text file. With it, it's still a text file, but the shell knows it contains instructions and will try to execute them when the script is launched.

The whole point of writing scripts is that they run, so the first basic step is to know how to let Linux know your script should be considered executable.

The `chmod` command lets us set file permissions. The execute permission can be set with the +x flag.

```
chmod +x script1.sh
```



```
dave@Ubuntu-22-04:~/htg$ chmod +x script1.sh
```

You'll need to do this to each of your scripts. Replace "script1.sh" with the name of your script.

1. What's That Strange First Line?

The first line of a script tells the shell which interpreter should be called to run that script. The first line must start with a shebang, "#!", also known as a hashbang. The "#!" tells the shell that this line contains the path and name of the interpreter that the script was written for.

This is important because if you've written a script to run in Bash, you don't want it to be interpreted by a different shell. There are likely to be incompatibilities. Bash—like most shells—has its own quirks of syntax and functionality that other shells won't have, or will have implemented differently.

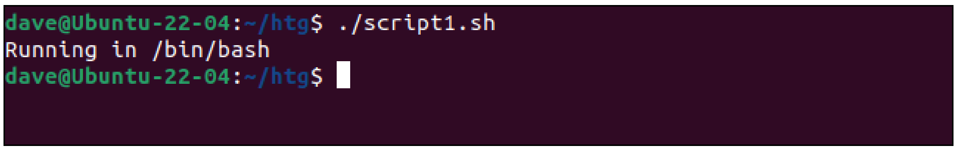
When you run a script, the current shell opens the script and determines which shell or interpreter should be used to execute that script. It then launches that shell and passes the script to it.

```
#!/bin/bash

echo Running in $SHELL
```

The first line of this script can be read as "Use the interpreter located at /bin/bash to run this script."

The only line in the script writes the value held in the `$SHELL` environmental variable to the terminal screen. This confirms that Bash was used to execute the script.

```
./script1.sh
```

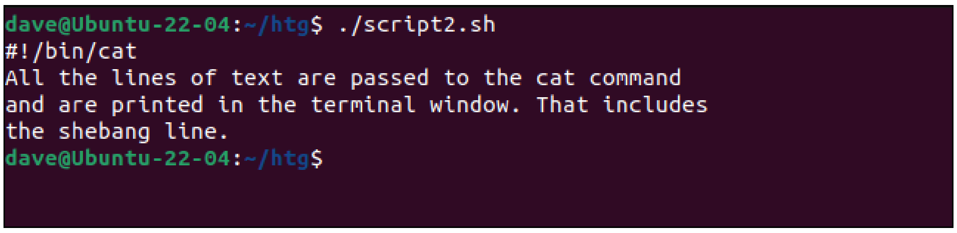


```
dave@Ubuntu-22-04:~/htg$ ./script1.sh
Running in /bin/bash
dave@Ubuntu-22-04:~/htg$
```

As a bit of a parlor trick, we can demonstrate that the script is passed to any interpreter we select.

```
#!/bin/cat
All the lines of text are passed to the cat command
and are printed in the terminal window. That includes
the shebang line.
```

```
script2.sh
```



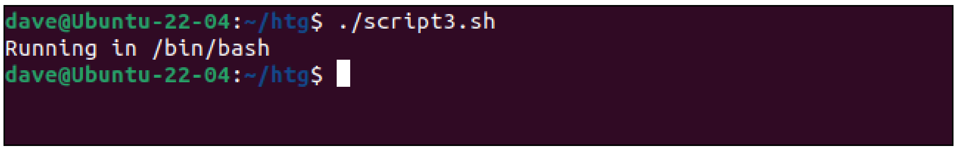
```
dave@Ubuntu-22-04:~/htg$ ./script2.sh
#!/bin/cat
All the lines of text are passed to the cat command
and are printed in the terminal window. That includes
the shebang line.
dave@Ubuntu-22-04:~/htg$
```

This script is launched by the current shell and passed to [the `cat` command]. The `cat` command "runs" the script.

Writing your shebangs like this makes an assumption that you know where the shell or other interpreter is located on the target machine. And 99% of the time, that's fine. But some people like to hedge their bets and write their shebangs like this:

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

echo Running in $SHELL
```

```
script3.sh
```



```
dave@Ubuntu-22-04:~/htg$ ./script3.sh
Running in /bin/bash
dave@Ubuntu-22-04:~/htg$
```

When the script is launched the shell *searches* for the location of the named shell. If the shell happens to be in a non-standard location, this type of approach can avoid "bad interpreter" errors.

Don't Listen, He's Lying!

In Linux, there's always more than one way to skin a cat or prove an author wrong. To be completely factual, there is a way to run scripts without a shebang, and without making them executable.

If you launch the shell that you want to execute the script and pass the script as a command line parameter, the shell will launch and run the script— whether it is executable or not. Because you choose the shell on the command line, there's no need for a shebang.

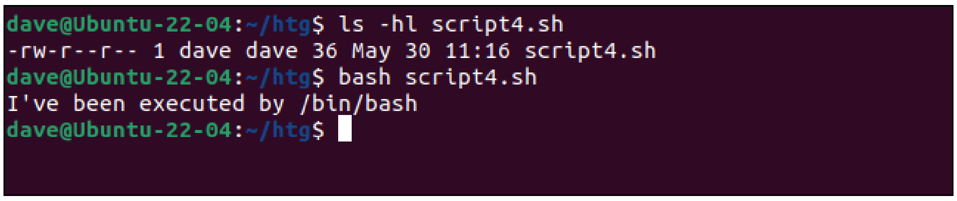
This is the entire script:

```
echo "I've been executed by" $SHELL
```

We'll use `ls` to see that the script really isn't executable and launch Bash with the name of the script:

```
ls
```

```
bash script4.sh
```



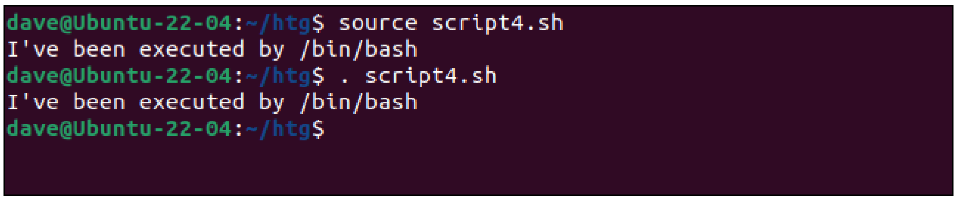
```
dave@Ubuntu-22-04:~/htg$ ls -hl script4.sh
-rw-r--r-- 1 dave dave 36 May 30 11:16 script4.sh
dave@Ubuntu-22-04:~/htg$ bash script4.sh
I've been executed by /bin/bash
dave@Ubuntu-22-04:~/htg$
```

There's also a way to have a script run by the *current* shell, not a shell launched specifically to execute the script. If you use the `source` command, which can be abbreviated to a single period "`.`", your script is executed by your current shell.

So, to run a script without a shebang, without the executable file permission, and without launching another shell, you can use either of these commands:

```
source script4.sh
```

```
. script4.sh
```



```
dave@Ubuntu-22-04:~/htg$ source script4.sh
I've been executed by /bin/bash
dave@Ubuntu-22-04:~/htg$ . script4.sh
I've been executed by /bin/bash
dave@Ubuntu-22-04:~/htg$
```

Although this is possible, it isn't recommended as a general solution. There are drawbacks.

If a script doesn't contain a shebang, you can't tell which shell it was written for. Are you going to remember in a year's time? And without the executable permission being set on the script, the `ls` command won't identify it as an executable file, nor will it use color to distinguish the script from plain text files.

RELATED: ***Command Lines: Why Do People Still Bother With Them?***

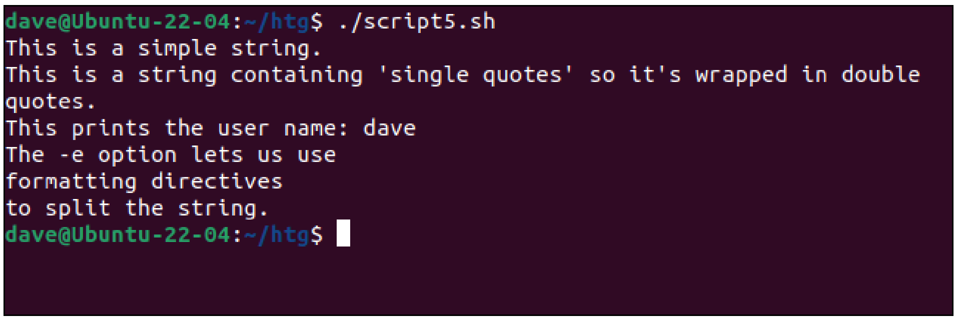
2. Printing Text

Writing text to the terminal is a common requirement. A bit of visual feedback goes a long way.

For simple messages, the `echo` command will suffice. It allows some formatting of the text and lets you work with variables too.

```
#!/bin/bash

echo This is a simple string.
echo "This is a string containing 'single quotes' so it's wr
echo "This prints the user name:" $USER
echo -e "The -e option lets us use\nformatting directives\nt
```

```
./script5.sh
```



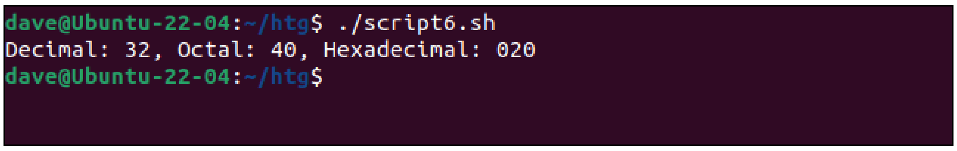
```
dave@Ubuntu-22-04:~/htg$ ./script5.sh
This is a simple string.
This is a string containing 'single quotes' so it's wrapped in double
quotes.
This prints the user name: dave
The -e option lets us use
formatting directives
to split the string.
dave@Ubuntu-22-04:~/htg$
```

The `printf` command gives us more flexibility and better formatting capabilities including number conversion.

This script prints the same number using three different numerical bases. The hexadecimal version is also formatted to print in uppercase, with leading zeroes and a width of three digits.

```
#!/bin/bash

printf "Decimal: %d, Octal: %o, Hexadecimal: %03X\n" 32 32 3
```

```
./script6.sh
```



```
dave@Ubuntu-22-04:~/htg$ ./script6.sh
Decimal: 32, Octal: 40, Hexadecimal: 020
dave@Ubuntu-22-04:~/htg$
```

Note that unlike with `echo`, you must tell `printf` to start a new line with the “`\n`” token.

3. Creating and Using Variables

Variables allow you to store values inside your program and to manipulate and use them. You can create your own variables or use environment variables for system values.

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

```
millennium_text="Years since the millennium:"

current_time=$( date '+%H:%M:%S' )
todays_date=$( date '+%F' )
year=$( date '+%Y' )

echo "Current time:" $current_time
echo "Today's date:" $todays_date

years_since_Y2K=$(( year - 2000 ))

echo $millennium_text $years_since_Y2K
```

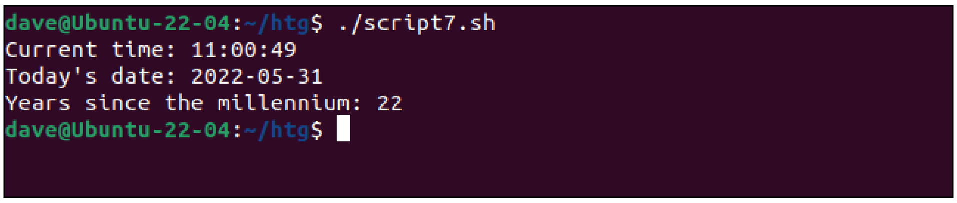
This script creates a string variable called `millennium_text`. It holds a line of text.

It then creates three numerical variables.

- The `current_time` variable is initialized to the time the script is executed.
- The `todays_date` variable is set to the date on which the script is run.
- The `year` variable holds the current year.

To access the *value* stored in a variable, precede its name with a dollar sign "$."

```
./script7.sh
```



```
dave@Ubuntu-22-04:~/htg$ ./script7.sh
Current time: 11:00:49
Today's date: 2022-05-31
Years since the millennium: 22
dave@Ubuntu-22-04:~/htg$
```

The script prints the time and date, then calculates how many years have passed since the millennium, and stores this in the `years_since_Y2K` variable.

Finally, it prints the string contained in the `millennium_text` variable and the numerical value stored in the `years_since_Y2K`.

RELATED: ***How to Work with Variables in Bash***

4. Handling User Input

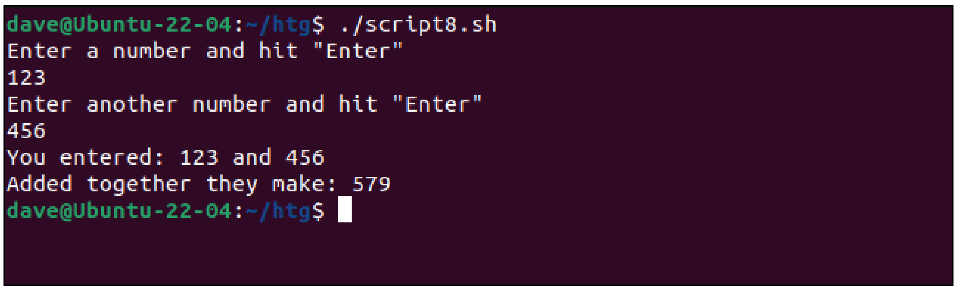
To allow a user to enter a value that the script will use, you need to be able to capture the user's keyboard input. The Bash `read` command allows ut to do just that. Here's a simple example.

```
#!/bin/bash

echo "Enter a number and hit \"Enter\""
read user_number1;
echo "Enter another number and hit \"Enter\""
read user_number2;

printf "You entered: %d and %d\n" $user_number1 $user_number
printf "Added together they make: %d\n" $(( user_number1 + u
```

The script prompts for two numbers. They are read from the keyboard and stored in two variables, `user_number1` and `user_number2` .

The script prints the numbers to the terminal window, adds them together, and prints the total.

```
./script8.sh
```



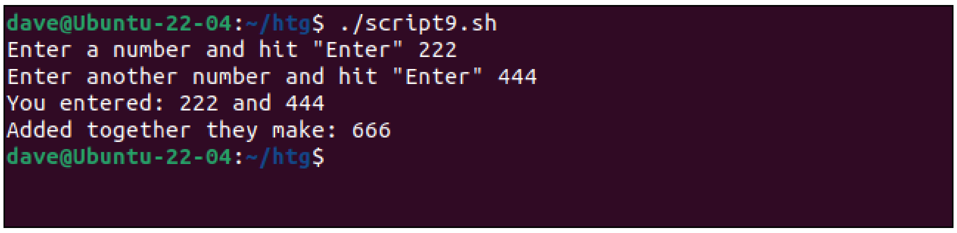
```
dave@Ubuntu-22-04:~/htg$ ./script8.sh
Enter a number and hit "Enter"
123
Enter another number and hit "Enter"
456
You entered: 123 and 456
Added together they make: 579
dave@Ubuntu-22-04:~/htg$
```

We can combine the prompts into the `read` commands using the `-p` (prompt) option.

```
#!/bin/bash

read -p "Enter a number and hit \"Enter\" " user_number1;
read -p "Enter another number and hit \"Enter\" " user_numbe

printf "You entered: %d and %d\n" $user_number1 $user_number
printf "Added together they make: %d\n" $(( user_number1 + u
```

This makes things neater and easier to read. Scripts that are easy to read are also easier to debug.

```
./script9.sh
```



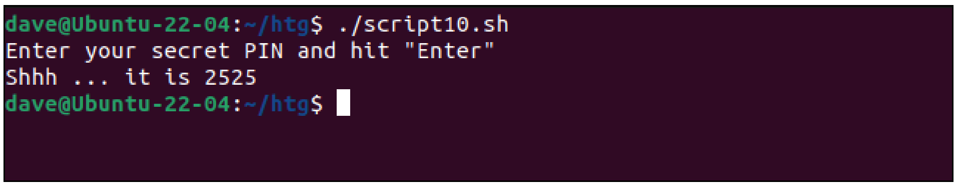
```
dave@Ubuntu-22-04:~/htg$ ./script9.sh
Enter a number and hit "Enter" 222
Enter another number and hit "Enter" 444
You entered: 222 and 444
Added together they make: 666
dave@Ubuntu-22-04:~/htg$
```

The script behaves slightly differently now. The user input is on the same line as the prompt.

To capture keyboard input without having it echoed to the terminal window, use the `-s` (silent) option.

```
#!/bin/bash

read -s -p "Enter your secret PIN and hit \"Enter\" " secret

printf "\nShhh ... it is %d\n" $secret_PIN
```

```
./script10.sh
```



```
dave@Ubuntu-22-04:~/htg$ ./script10.sh
Enter your secret PIN and hit "Enter"
Shhh ... it is 2525
dave@Ubuntu-22-04:~/htg$
```

The input value is captured and stored in a variable called `secret_PIN`, but it isn't echoed to the screen when the user *types it*. What you do with it after that is up to you.

5. Accepting Parameters

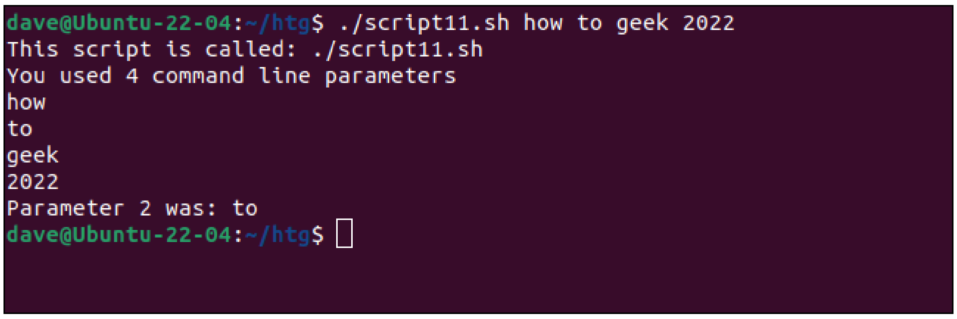
Sometimes it is more convenient to accept user input as command line parameters than to have a script sit waiting for input. Passing values to a script is easy. They can be referenced inside the script as if they were any other variable.

The first parameter becomes variable `$1`, the second parameter becomes variable `$2`, and so on. Variable `$0` always holds the name of the script, and variable `$#` holds the number of parameters that were provided on the command line. Variable `$@` is a string that contains all of the command line parameters.

```
#!/bin/bash

printf "This script is called: %s\n" $0
printf "You used %d command line parameters\n" $#

# loop through the variables
for param in "$@"; do
  echo "$param"
done

echo "Parameter 2 was:" $2
```

This script uses `$0` and `$#` to print some information. then uses `?@` to loop through all of the command line parameters. It uses `$2` to show how to access a single, particular parameter value.

```
./script11.sh
```



```
dave@Ubuntu-22-04:~/htg$ ./script11.sh how to geek 2022
This script is called: ./script11.sh
You used 4 command line parameters
how
to
geek
2022
Parameter 2 was: to
dave@Ubuntu-22-04:~/htg$ 
```

Wrapping several words in quotation marks “”” combines them into a single parameter.

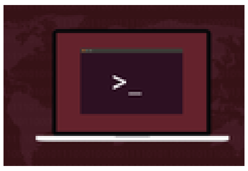
6. Reading Data From Files

Knowing how to read data from a file is a great skill to have. We can do this in Bash [with a while loop](#).

```
#!/bin/bash

LineCount=0

while IFS='' read -r LinefromFile || [[ -n "${LinefromFile}"

  ((LineCount++))
  echo "Reading line $LineCount: ${LinefromFile}"
```

```
done < "$1"
```

We're passing the name of the file we want the script to process as a command line parameter. It'll be the only parameter, so inside the script `$1` will hold the filename. We're redirecting that file into the `while` loop.

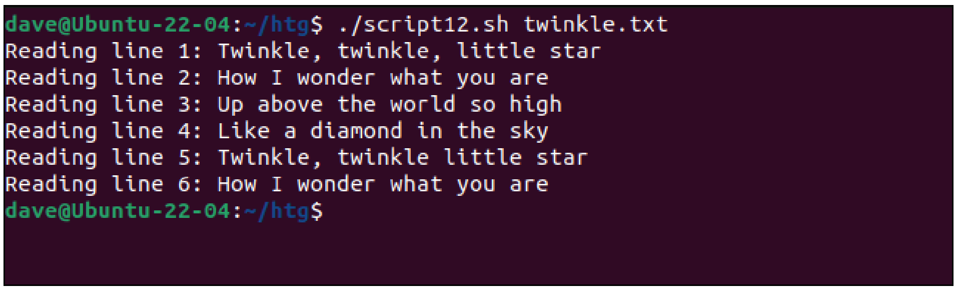


RELATED
How to Process a File Line by Line in a Linux Bash Script

The `while` loop sets the internal field separator to an empty string, using the `IFS=''` assignment. This prevents the `read` command from splitting lines at whitespace. Only the carriage return at the end of a line is considered to be the true end of the line.

The `[[ -n "${LinefromFile}" ]]` clause caters for the possibility that the last line in the file doesn't end with a carriage return. Even if it doesn't, that last line will be handled correctly and treated as a regular POSIX-compliant line.

```
./script12.sh twinkle.txt
```



```
dave@Ubuntu-22-04:~/htg$ ./script12.sh twinkle.txt
Reading line 1: Twinkle, twinkle, little star
Reading line 2: How I wonder what you are
Reading line 3: Up above the world so high
Reading line 4: Like a diamond in the sky
Reading line 5: Twinkle, twinkle little star
Reading line 6: How I wonder what you are
dave@Ubuntu-22-04:~/htg$
```

7. Using Conditional Tests

If you want your script to perform different actions for different conditions, you need to perform conditional tests. The double-bracket test syntax delivers an—at first—overwhelming number of options.

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

```
price=$1

if [[ price -ge 15 ]];
then
  echo "Too expensive."
else
  echo "Buy it!"
fi
```

Bash provides a whole set of comparison operators that let you determine things such as whether a file exists, if you can read from it, if you can write to it, and whether a directory exists.

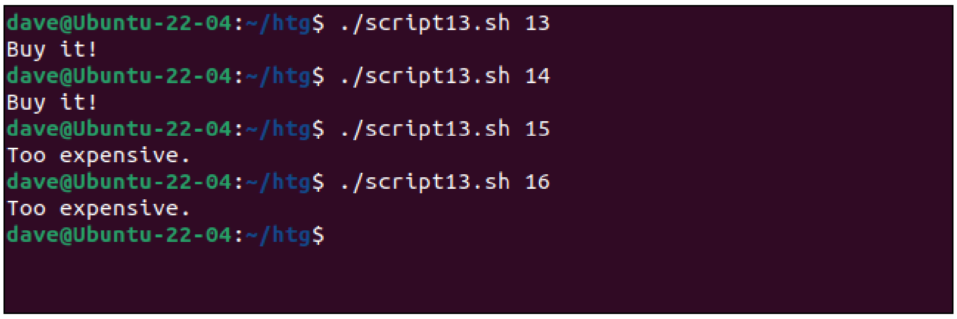
It also has numerical tests for equals `-qe`, greater than `-gt`, less than or equal `-le`, and so on, although you can also use the familiar `==`, `>=`, `<=` notation.

```
./script13.sh 13
```

```
./script13.sh 14
```

```
./script13.sh 15
```

```
./script13.sh 16
```



```
dave@Ubuntu-22-04:~/htg$ ./script13.sh 13
Buy it!
dave@Ubuntu-22-04:~/htg$ ./script13.sh 14
Buy it!
dave@Ubuntu-22-04:~/htg$ ./script13.sh 15
Too expensive.
dave@Ubuntu-22-04:~/htg$ ./script13.sh 16
Too expensive.
dave@Ubuntu-22-04:~/htg$
```

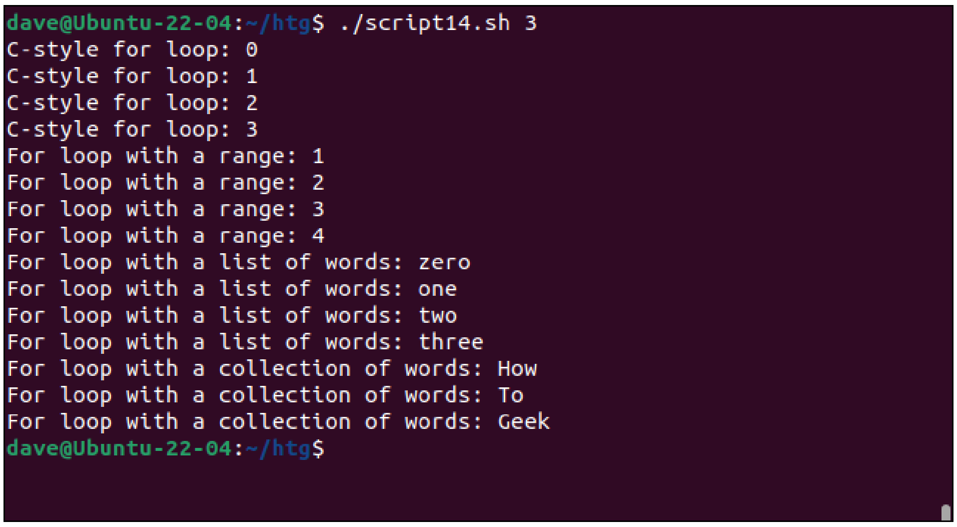
8. The Power of for Loops

Repeating actions over and over is best accomplished using loops. A `for` loop lets you run a loop a number of times. This might be up to a particular number, or it might be until the loop has worked its way through a list of items.

```
#!/bin/bash

for (( i=0; i<=$1; i++ ))
do
  echo "C-style for loop:" $i
done

for i in {1..4}
do
  echo "For loop with a range:" $i
done

for i in "zero" "one" "two" "three"
do
  echo "For loop with a list of words:" $i
done

website="How To Geek"

for i in $website
do
  echo "For loop with a collection of words:" $i
done
```

All of these loops are `for` loops, but they're working with different types of loop statements and data.

```
./script14.sh 3
```



```
dave@Ubuntu-22-04:~/htg$ ./script14.sh 3
C-style for loop: 0
C-style for loop: 1
C-style for loop: 2
C-style for loop: 3
For loop with a range: 1
For loop with a range: 2
For loop with a range: 3
For loop with a range: 4
For loop with a list of words: zero
For loop with a list of words: one
For loop with a list of words: two
For loop with a list of words: three
For loop with a collection of words: How
For loop with a collection of words: To
For loop with a collection of words: Geek
dave@Ubuntu-22-04:~/htg$
```

The first loop is a classic C-style `for` loop. The loop counter `i` is initialized to zero, and incremented with each cycle of the loop. While the value of `i` is less than or equal to the value held in `$1`, the loop will continue to run.

The second loop works through the range of numbers from 1 to 4. The third loop works through a list of words. While there are more words to process, the loop keeps repeating.

The last loop works through the list of words in a string variable.

9. Functions

Functions allow you to encapsulate sections of code into named routines that can be called from anywhere within your script.

Suppose we wanted our script that reads lines from a file to do some sort of processing on each line. It would be convenient to have that code contained within a function.

```
#!/bin/bash

LineCount=0

function count_words() {
  printf "%d words in line %d\n" $(echo $1 | wc -w) $2
}

while IFS='' read -r LinefromFile || [[ -n "${LinefromFile}"

  ((LineCount++))
  count_words "$LinefromFile" $LineCount

done < "$1"

count_words "This isn't in the loop" 99
```

We've modified our file reading program by adding a function called `count_words`. It is defined *before* we need to use it.

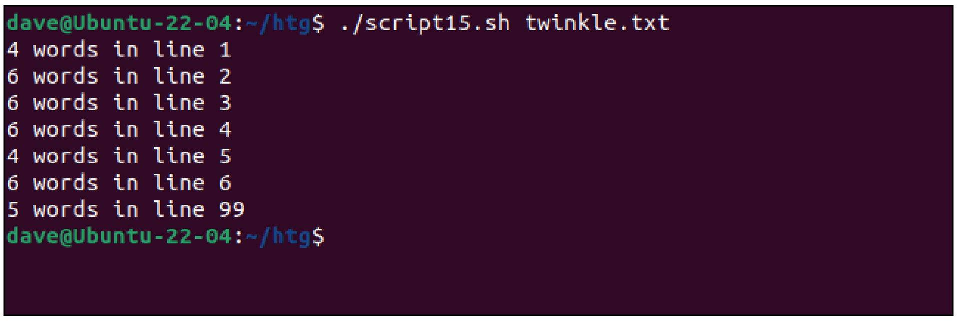
The function definition starts with the word `function`. This is followed by a unique name for our function followed by parentheses "`()`." The body of the function is contained within curly brackets "{}."

The function definition doesn't cause any code to be executed. Nothing in the function is run until the function is called.

The `count_words` function prints the number of words in a line of text, and the line number. These two parameters are passed into the function just like parameters are passed into a script. The first parameter becomes *function* variable `$1`, and the second parameter becomes function variable `$2`, and so on.

The `while` loop reads each line from the file and passes it to the `count_words` function, together with the line number. And just to show we can call the function from different places within the script, we call it once more outside of the `while` loop.

```
./script15.sh twinkle.txt
```



```
dave@Ubuntu-22-04:~/htg$ ./script15.sh twinkle.txt
4 words in line 1
6 words in line 2
6 words in line 3
6 words in line 4
4 words in line 5
6 words in line 6
5 words in line 99
dave@Ubuntu-22-04:~/htg$
```

Don’t Fear the Learning Curve

Scripting is rewarding and useful, but tough to get into. Once you get some re-usable techniques under your belt you’ll be able to write worthwhile scripts relatively easily. Then you can look into more advanced functionality.

Walk before you can run, and take time to enjoy the journey.

RELATED: ***10 Basic Linux Commands for Beginners***

DAVE MCKAY

Dave McKay first used computers when punched paper tape was in vogue, and he has been programming ever since. After over 30 years in the IT industry, he is now a full-time technology journalist. During his career, he has worked as a freelance programmer, manager of an international software development team, an IT services project manager, and, most recently, as a Data Protection Officer. His writing has been published by howtogeek.com, cloudsavvyit.com, itenterpriser.com, and opensource.com. Dave is a Linux evangelist and open source advocate. **READ FULL BIO »**