Linux Shell Scripting for Beginners

This tutorial is an introduction into shell scripting. We will cover the basics of the shell, parameters, return values and redirection. We will also cover variables, functions, if statements and loops.

To complete this tutorial, you will need access to a running Linux distribution, or 'distro' for short. There are a number of Linux 'distros'. If you do not already have access to a Linux system, I have a number of VirtualBox tutorials where I demonstrate the installation of CentOS 7 and Ubuntu 22 as virtual machines, accessible **here**.

I also have a few tutorials where I demonstrate the creation of AWS compute instances, RHEL 8 and Ubuntu 20, accessible **here**. Keep in mind that you will need to create an AWS account to be able to create a compute instance. If you do not have an AWS account, my tutorial **Create AWS Free Tier Account** is accessible **here.**

If you prefer, you can use a CentOS 7 or Ubuntu 22 VM instead. The choice is yours.

For this tutorial, I will be using both my RHEL 8, and Ubuntu 20, AWS compute instances, and I will be providing screenshots from my Ubuntu 20 instance. When I encounter a difference in output, I will include the RHEL 8 screenshot.

In this tutorial, I will go through the following:

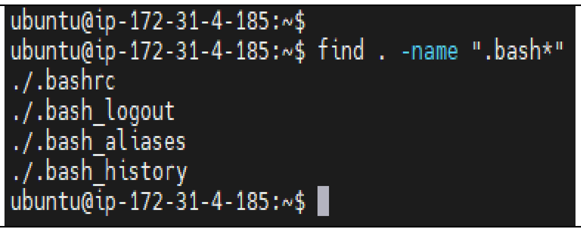
Now that you have a running Linux system, are logged in and have access to the command line, we can begin.

Introduction to Shell Scripting

A shell is a command line interpreter which provides the user interface for terminal windows. It can also be used to run scripts. For example typing: `find . -name ".bash*"` at the command line accomplishes the same thing as executing a script file containing the lines:

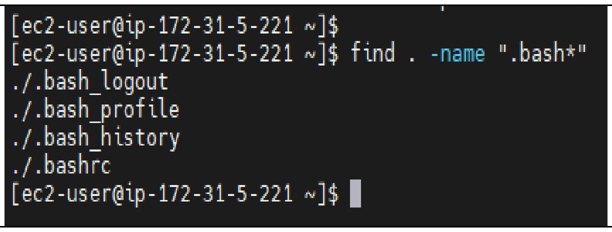
```
#!/bin/bash
find . -name ".bash*"
```

Ubuntu 20



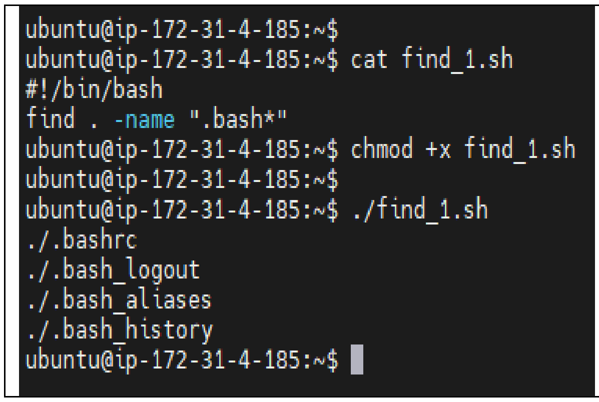
```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ find . -name ".bash*"
./.bashrc
./.bash_logout
./.bash_aliases
./.bash_history
ubuntu@ip-172-31-4-185:~$
```

RHEL 8

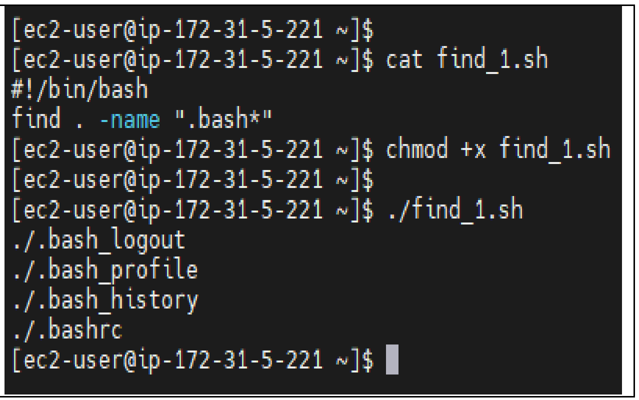


```
[ec2-user@ip-172-31-5-221 ~]$
[ec2-user@ip-172-31-5-221 ~]$ find . -name ".bash*"
./.bash_logout
./.bash_profile
./.bash_history
./.bashrc
[ec2-user@ip-172-31-5-221 ~]$
```

The first line of the script, that starts with **#! (**known as **shebang)**, contains the full path of the command interpreter (in this case **/bin/bash**)

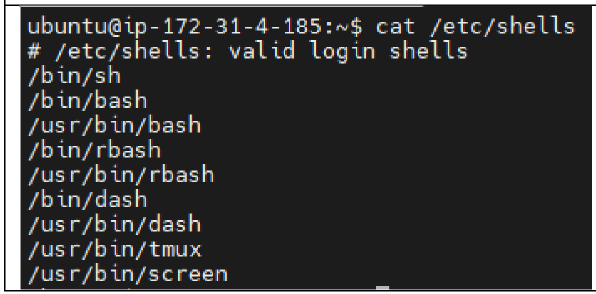
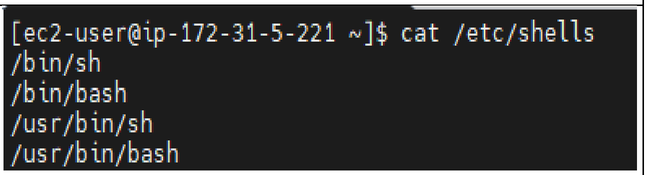


```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat find_1.sh
#!/bin/bash
find . -name ".bash*"
ubuntu@ip-172-31-4-185:~$ chmod +x find_1.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./find_1.sh
./.bashrc
./.bash_logout
./.bash_aliases
./.bash_history
ubuntu@ip-172-31-4-185:~$
```



```
[ec2-user@ip-172-31-5-221 ~]$
[ec2-user@ip-172-31-5-221 ~]$ cat find_1.sh
#!/bin/bash
find . -name ".bash*"
[ec2-user@ip-172-31-5-221 ~]$ chmod +x find_1.sh
[ec2-user@ip-172-31-5-221 ~]$
[ec2-user@ip-172-31-5-221 ~]$ ./find_1.sh
./.bash_logout
./.bash_profile
./.bash_history
./.bashrc
[ec2-user@ip-172-31-5-221 ~]$
```

The command interpreter is tasked with executing statements that follow it in the script. Linux provides a wide choice of shells; exactly what is available on the system is listed in **/etc/shells**

Ubuntu 20	RHEL 8
	

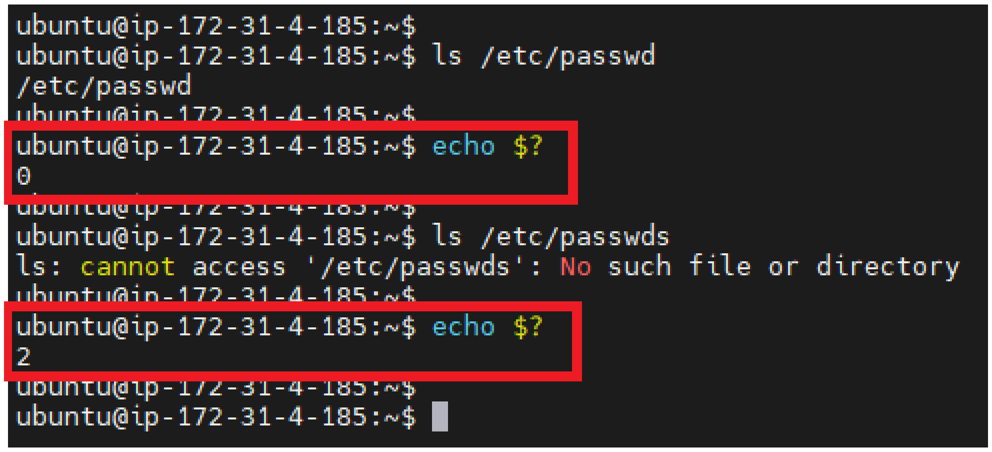
```
ubuntu@ip-172-31-4-185:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/bin/tmux
/usr/bin/screen
```

```
[ec2-user@ip-172-31-5-221 ~]$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
```

Return Values

All shell scripts generate a return value upon finishing execution. The return value can also be set with the **exit** statement in a shell script. Return values help determine how the script terminated. As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value.

We will demonstrate both success and failure completion by executing the **ls** command on a file that exists and one that doesn't. The return value is stored in the environment variable **$?**



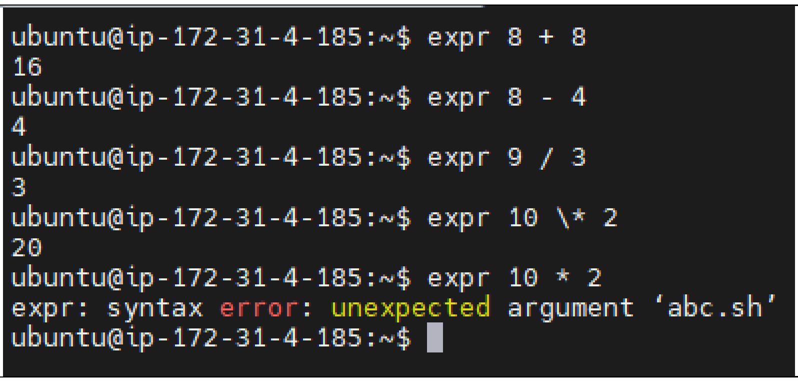
```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls /etc/passwd
/etc/passwd
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ echo $?
0
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ls /etc/passwds
ls: cannot access '/etc/passwds': No such file or directory
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ echo $?
2
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$
```

Arithmetic Expressions

Arithmetic expressions can be evaluated in the following three ways:

Using the expr utility: **expr** is a standard but deprecated program. The syntax is as follows:

```
$ expr 8 + 8
$ expr 8 - 4
$ expr 9 / 3
$ expr 10 \* 2
```

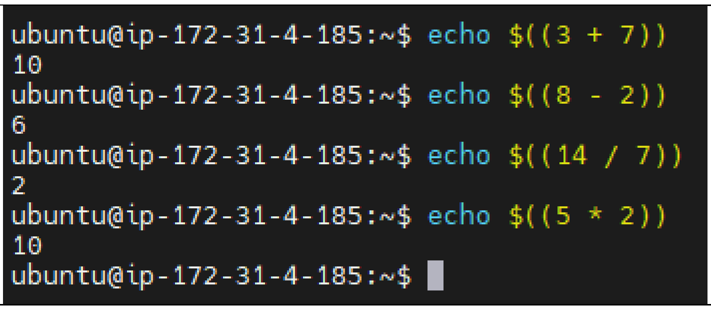


```
ubuntu@ip-172-31-4-185:~$ expr 8 + 8
16
ubuntu@ip-172-31-4-185:~$ expr 8 - 4
4
ubuntu@ip-172-31-4-185:~$ expr 9 / 3
3
ubuntu@ip-172-31-4-185:~$ expr 10 \* 2
20
ubuntu@ip-172-31-4-185:~$ expr 10 * 2
expr: syntax error: unexpected argument ‘abc.sh’
ubuntu@ip-172-31-4-185:~$
```

Note above that since the multiplication operator ***** is considered a special character, I had to escape it with a backslash \. Otherwise, an error is generated.

Using the **$((...))** syntax: This is the built-in shell format. The syntax is as follows:

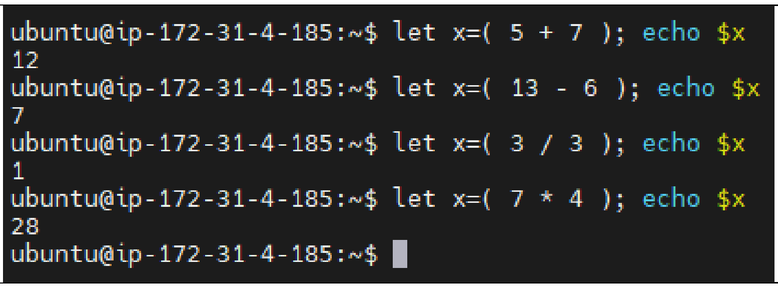
```
$ echo $((3 + 7))
$ echo $((8 - 2))
$ echo $((14 / 7))
$ echo $((5 * 2))
```



```
ubuntu@ip-172-31-4-185:~$ echo $((3 + 7))
10
ubuntu@ip-172-31-4-185:~$ echo $((8 - 2))
6
ubuntu@ip-172-31-4-185:~$ echo $((14 / 7))
2
ubuntu@ip-172-31-4-185:~$ echo $((5 * 2))
10
ubuntu@ip-172-31-4-185:~$
```

Using the built-in shell command **let**. The syntax is as follows:

```
$ let x=( 5 + 7 ); echo $x
$ let x=( 13 - 6 ); echo $x
$ let x=( 3 / 3 ); echo $x
$ let x=( 7 * 4 ); echo $x
```



```
ubuntu@ip-172-31-4-185:~$ let x=( 5 + 7 ); echo $x
12
ubuntu@ip-172-31-4-185:~$ let x=( 13 - 6 ); echo $x
7
ubuntu@ip-172-31-4-185:~$ let x=( 3 / 3 ); echo $x
1
ubuntu@ip-172-31-4-185:~$ let x=( 7 * 4 ); echo $x
28
ubuntu@ip-172-31-4-185:~$
```

Boolean Expressions

Boolean expressions evaluate to either TRUE or FALSE, and results are obtained using the various Boolean operators listed in the table.

Operator	Operation	Meaning
&&	AND	The action will be performed only if both the conditions evaluate to true.
||	OR	The action will be performed if any one of the conditions evaluate to true.
!	NOT	The action will be performed only if the condition evaluates to false.

Note that if you have multiple conditions strung together with the && operator processing stops as soon as a condition evaluates to false.

For example if you have A && B && C and A is true but B is false, C will never be executed.

Likewise if you are using the || operator, processing stops as soon as anything is true.

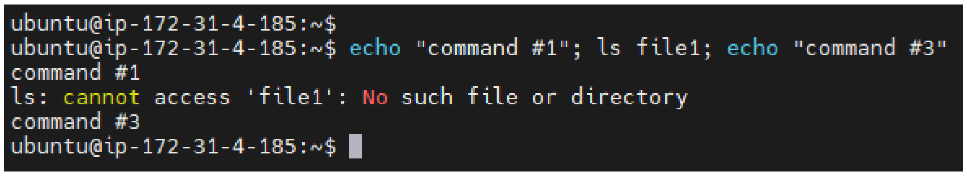
For example if you have A || B || C and A is false and B is true, you will also never execute C.

Putting Multiple Commands on a Single Line

Sometimes you may want to group multiple commands on a single line. The **;** (semicolon) character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.

The three commands in the following example will all execute even if the ones preceding them fail:

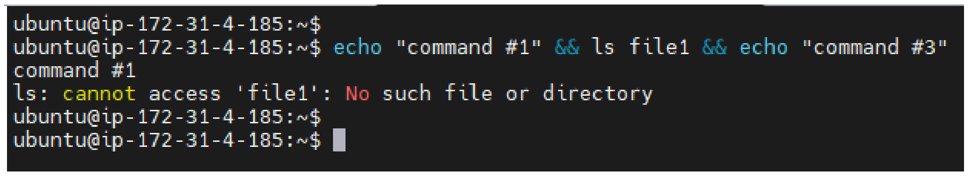
```
$ echo "command #1"; ls file1; echo "command #3"
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ echo "command #1"; ls file1; echo "command #3"
command #1
ls: cannot access 'file1': No such file or directory
command #3
ubuntu@ip-172-31-4-185:~$
```

However, you can abort subsequent commands if one fails using the && (and) operator:

```
$ echo "command #1" && ls file1 && echo "command #3"
```

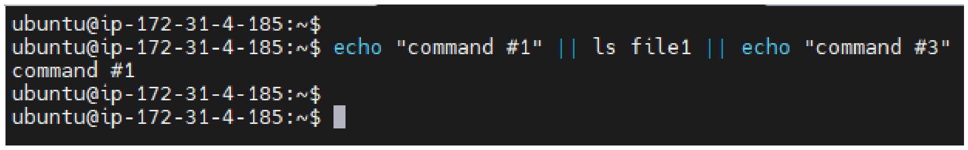


```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ echo "command #1" && ls file1 && echo "command #3"
command #1
ls: cannot access 'file1': No such file or directory
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$
```

Notice above that the third command is never executed due to the failure from the non-existent file.

A final option is to use the || (or) operator. As soon as something succeeds, execution is terminated.

```
$ echo "command #1" || ls file1 || echo "command #3"
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ echo "command #1" || ls file1 || echo "command #3"
command #1
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$
```

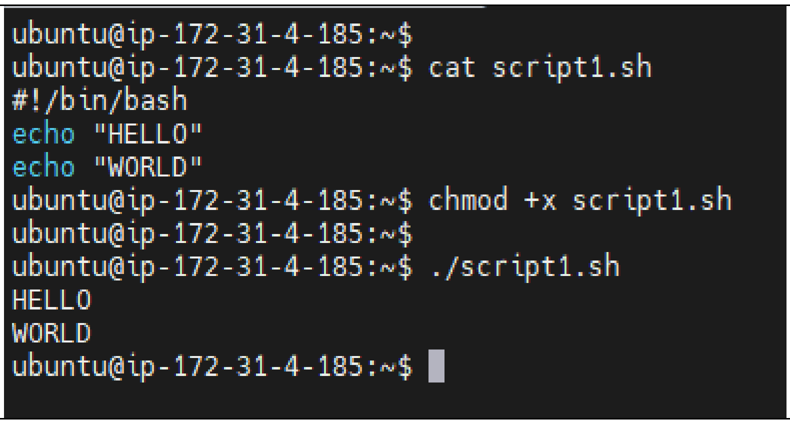
bash Scripting

We will now create a simple bash script named **script1.sh** that displays a two-line message on the screen. Then, make the file executable and, finally, execute the script.

```
#!/bin/bash
echo "HELLO"
echo "WORLD"
```

# make file executable for all users.

# execute the script.



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat script1.sh
#!/bin/bash
echo "HELLO"
echo "WORLD"
ubuntu@ip-172-31-4-185:~$ chmod +x script1.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./script1.sh
HELLO
WORLD
ubuntu@ip-172-31-4-185:~$
```

Script Parameters

Users often need to pass parameter values to a script. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers:

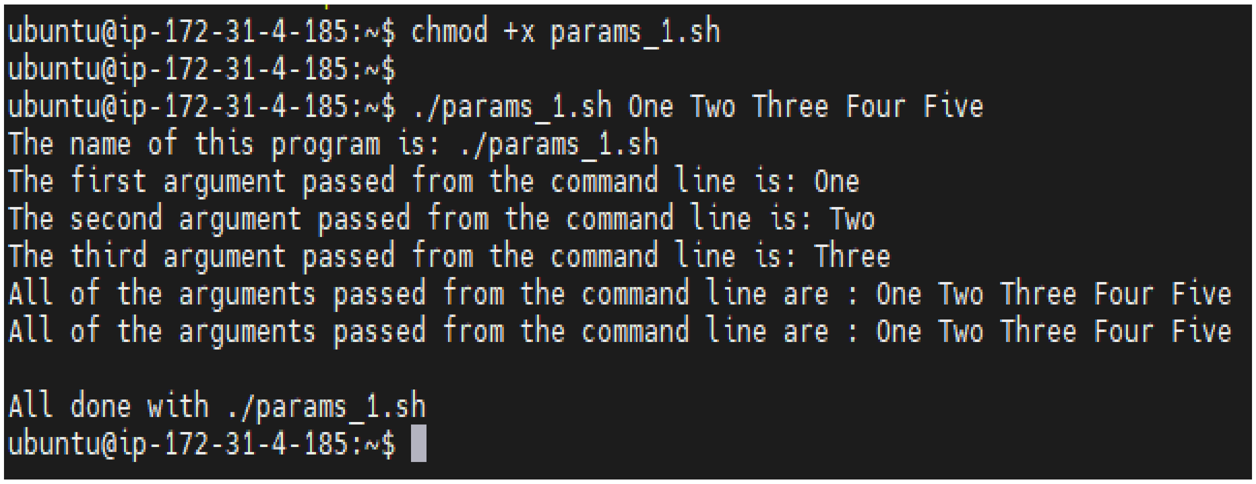
```
$ ./script.sh /tmp
$ ./script.sh 100 200
```

Within a script, the parameter or an argument is represented with a **$** and a number. The table lists some of these parameters.

Parameter	Meaning
$0	Script name
$1	First parameter
$2, $3, etc.	Second, third parameter, etc.
$*	All parameters - "$*" is the same as "$1 $2 $3 $4..." as one long string.
$@	All parameters - "$@" is the same as "$1" "$2" "$3" ..., all quoted separately.
$#	Number of arguments

This script, **params_1.sh**, will demonstrate the use of positional parameters. First, ensure that you make it executable. Then, call the script with at least 3 parameters.

```
#!/bin/bash
echo "The name of this program is: $0"
echo "The first argument passed from the command line is: $1"
echo "The second argument passed from the command line is: $2"
echo "The third argument passed from the command line is: $3"
echo "All of the arguments passed from the command line are : $*"
echo "All of the arguments passed from the command line are : $@"
echo
echo "All done with $0"
```



```
ubuntu@ip-172-31-4-185:~$ chmod +x params_1.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./params_1.sh One Two Three Four Five
The name of this program is: ./params_1.sh
The first argument passed from the command line is: One
The second argument passed from the command line is: Two
The third argument passed from the command line is: Three
All of the arguments passed from the command line are : One Two Three Four Five
All of the arguments passed from the command line are : One Two Three Four Five

All done with ./params_1.sh
ubuntu@ip-172-31-4-185:~$
```

Output Redirection

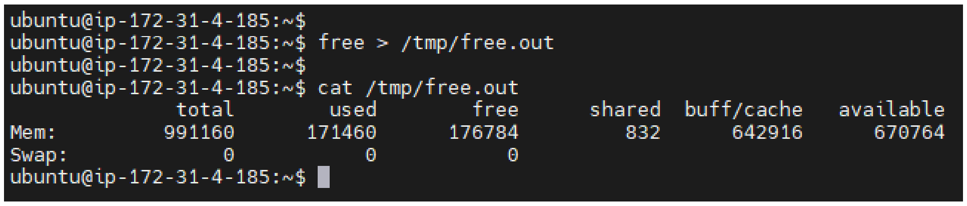
Most operating systems accept input from the keyboard and display the output on the terminal. However, using output redirection operators (>, >>) we can send the output to a file, or direct input to a command with input redirection (<).

In UNIX/Linux, all programs that run are given three open file streams when they are started as listed in the table:

I/O Name	Abbreviation	File Descriptor
Standard Input	stdin	0
Standard Output	stdout	1
Standard Error	stderr	2

The **>** character is used to write output to a file. For example, the following command sends the output of the **free** command to the file **/tmp/free.out**:

```
$ free > /tmp/free.out
$ cat /tmp/free.out
```

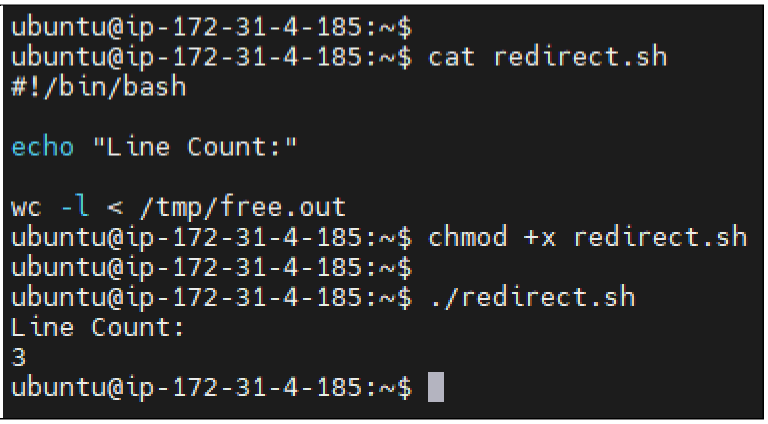


```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ free > /tmp/free.out
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat /tmp/free.out
              total        used        free      shared  buff/cache   available
Mem:         991160      171460      176784         832      642916      670764
Swap:             0           0           0
ubuntu@ip-172-31-4-185:~$
```

Two characters (**>>**) will append output to a file if it exists, and create the file if it does not already exist.

Just as the output can be redirected to a file, the input of a command can be read from a file. Input redirection uses the **<** character. Test by creating a script named **redirect.sh**:

```
#!/bin/bash
echo "Line count"
wc -l < /temp/free.out

# make it executable
$ chmod +x redirect.sh
# execute the script
$ ./redirect.sh
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat redirect.sh
#!/bin/bash

echo "Line Count:"

wc -l < /tmp/free.out
ubuntu@ip-172-31-4-185:~$ chmod +x redirect.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./redirect.sh
Line Count:
3
ubuntu@ip-172-31-4-185:~$
```

We can also redirect stdout & stderr to different files.

```
$ bash example.sh > logfile 2> errorfile
```

Or, redirect stdout & stderr to the same file

```
$ bash example.sh > logfile 2>&1
```

This is a shortcut for the above command.

```
$ bash example.sh &> logfile
```

Finally, if we have no need for the output generated by a command, or script, we can redirect the output to a special file called **/dev/null**. This file is also called the bit bucket or black hole.

```
$ find / -name "tmp" > /dev/null 2>&1
$ find / -name "tmp" &> /dev/null
$ ./example.sh &> /dev/null
```

Variables

Variables are storage locations that have a name. They are case sensitive and by convention variables are in uppercase. For example, VAR1="Value". We can reference the value of a shell variable by using a **$** (dollar sign) in front of the variable name, such as **$VAR1**.

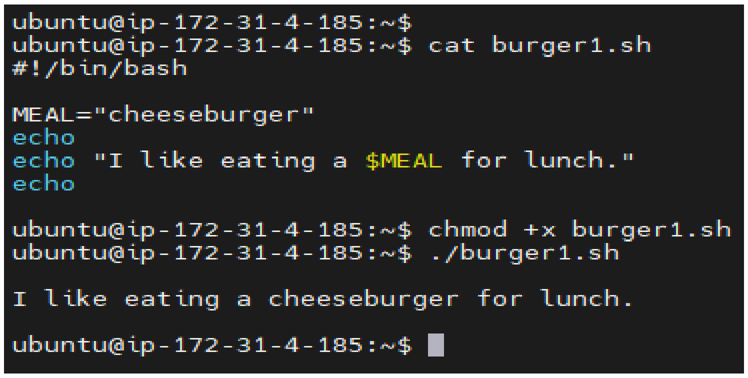
Variable names can contain only letters (a to z or A to Z), numbers (0 to 9) or underscores (_).

The following table lists both valid and invalid variable names:

Valid	Invalid
FIRST3LETTERS="ABC"	3LETTERS="ABC"
FIRST_THREE_LETTERS="ABC"	first-three-letters="ABC"
firstThreeLetters="ABC"	first@Three@Letters="ABC"

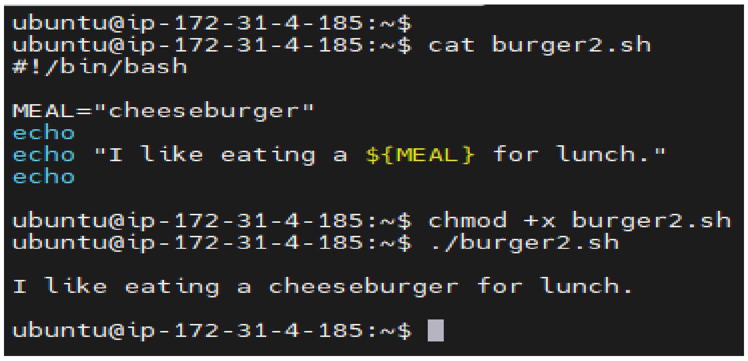
In the following example, we will use a variable and demonstrate when it is a good idea to use curly braces.

```
#!/bin/bash
MEAL="cheeseburger"
echo "I like eating a $MEAL for lunch.
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat burger1.sh
#!/bin/bash

MEAL="cheeseburger"
echo
echo "I like eating a $MEAL for lunch."
echo

ubuntu@ip-172-31-4-185:~$ chmod +x burger1.sh
ubuntu@ip-172-31-4-185:~$ ./burger1.sh

I like eating a cheeseburger for lunch.

ubuntu@ip-172-31-4-185:~$
```

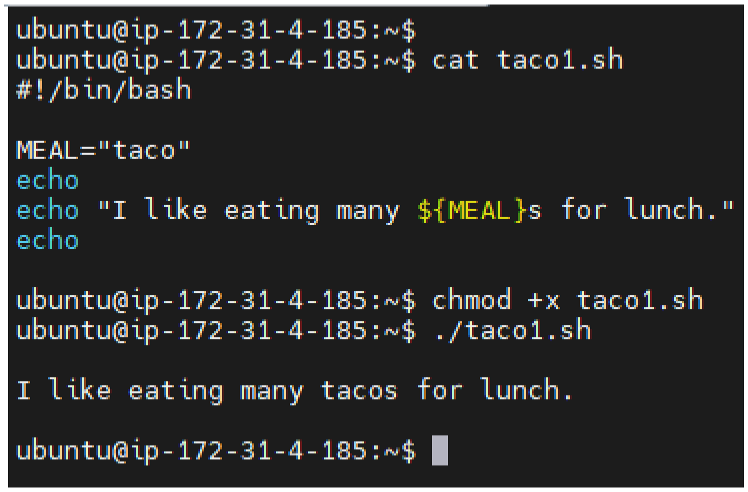
```
#!/bin/bash
MEAL="cheeseburger"
echo "I like eating a ${MEAL} for lunch.
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat burger2.sh
#!/bin/bash

MEAL="cheeseburger"
echo
echo "I like eating a ${MEAL} for lunch."
echo

ubuntu@ip-172-31-4-185:~$ chmod +x burger2.sh
ubuntu@ip-172-31-4-185:~$ ./burger2.sh

I like eating a cheeseburger for lunch.

ubuntu@ip-172-31-4-185:~$
```

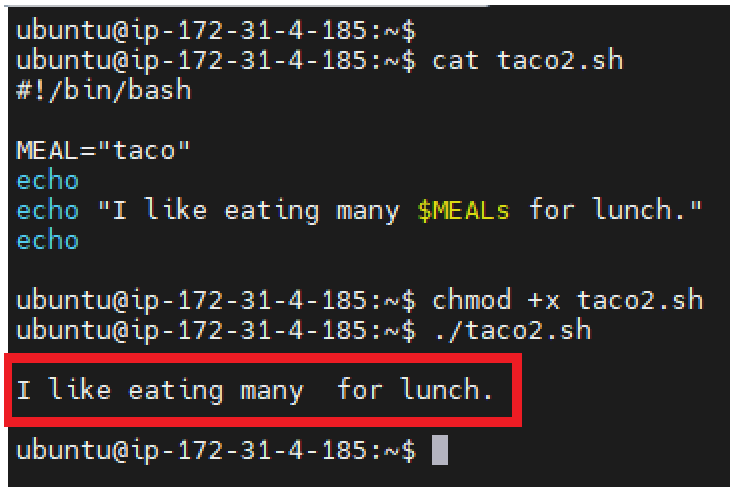
Notice the use of curly braces here. They tell the shell interpreter where the end of the variable name is. Although for this example, curly braces were not required. In the next example, they will be.

```
#!/bin/bash
MEAL="taco"
echo "I like eating many ${MEAL}s for lunch.
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat taco1.sh
#!/bin/bash

MEAL="taco"
echo
echo "I like eating many ${MEAL}s for lunch."
echo

ubuntu@ip-172-31-4-185:~$ chmod +x taco1.sh
ubuntu@ip-172-31-4-185:~$ ./taco1.sh

I like eating many tacos for lunch.

ubuntu@ip-172-31-4-185:~$
```

```
#!/bin/bash
MEAL="taco"
echo "I like eating a $MEALs for lunch.
```

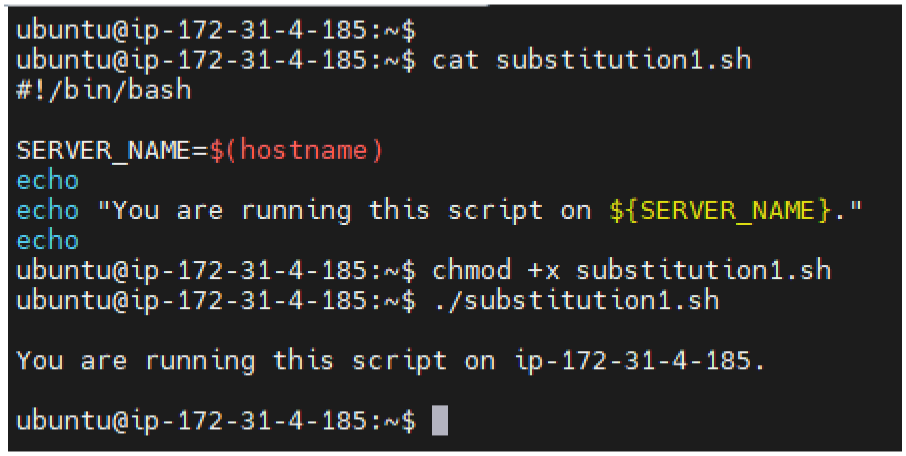


```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat taco2.sh
#!/bin/bash

MEAL="taco"
echo
echo "I like eating many $MEALs for lunch."
echo

ubuntu@ip-172-31-4-185:~$ chmod +x taco2.sh
ubuntu@ip-172-31-4-185:~$ ./taco2.sh

I like eating many  for lunch.

ubuntu@ip-172-31-4-185:~$
```

Notice that, in the second instance, the interpreter thought I was referring to a variable named **$MEALs**.

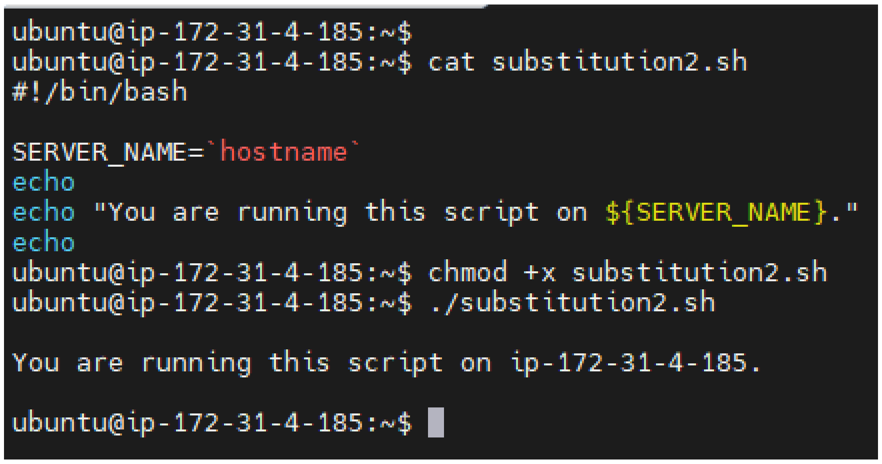
We can also use command substitution to assign command output to a variable. There are two methods of doing this. The older way was to use ` (backticks). The newer, and recommended way is to use **$( )**.

```
#!/bin/bash
SERVER_NAME=$(hostname)
echo "You are running this script on ${SERVER_NAME}."
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat substitution1.sh
#!/bin/bash

SERVER_NAME=$(hostname)
echo
echo "You are running this script on ${SERVER_NAME}."
echo
ubuntu@ip-172-31-4-185:~$ chmod +x substitution1.sh
ubuntu@ip-172-31-4-185:~$ ./substitution1.sh

You are running this script on ip-172-31-4-185.

ubuntu@ip-172-31-4-185:~$
```

```
#!/bin/bash
SERVER_NAME=`hostname`
echo "You are running this script on ${SERVER_NAME}."
```



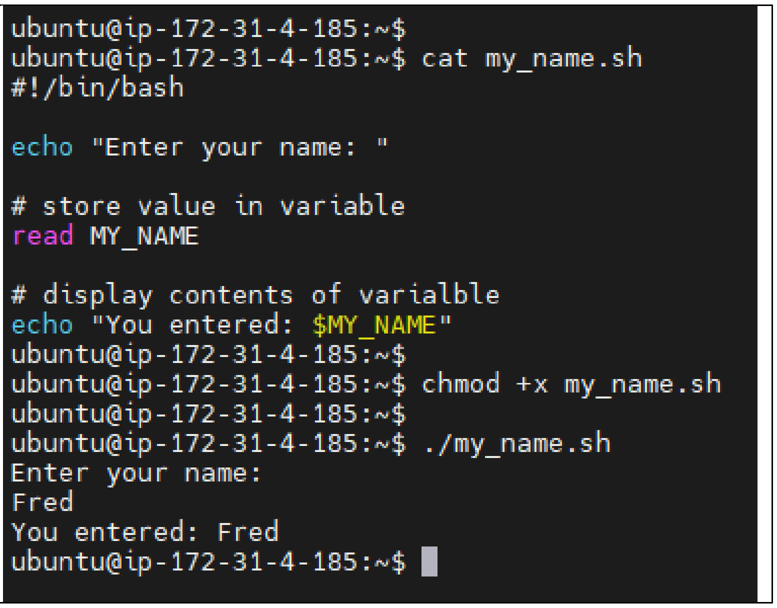
```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat substitution2.sh
#!/bin/bash

SERVER_NAME=`hostname`
echo
echo "You are running this script on ${SERVER_NAME}."
echo
ubuntu@ip-172-31-4-185:~$ chmod +x substitution2.sh
ubuntu@ip-172-31-4-185:~$ ./substitution2.sh

You are running this script on ip-172-31-4-185.

ubuntu@ip-172-31-4-185:~$
```

Now, we will create a more interactive example using a bash script. The user will be prompted to enter a value, which is then displayed on the screen. The value is stored in a temporary variable, **MY_NAME**. Create a script named **my_name.sh**.

```
#!/bin/bash
echo "Enter your name: "
# store value in variable
read MY_NAME
# display contents of MY_NAME
echo "You entered: $MY_NAME"
```

# make file executable for all users.

# execute the script.
# When prompted, I input a name and hit **Enter**.
# The name I entered, **Fred**, was displayed onscreen.



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat my_name.sh
#!/bin/bash

echo "Enter your name: "

# store value in variable
read MY_NAME

# display contents of varialble
echo "You entered: $MY_NAME"
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ chmod +x my_name.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./my_name.sh
Enter your name:
Fred
You entered: Fred
ubuntu@ip-172-31-4-185:~$
```

Functions

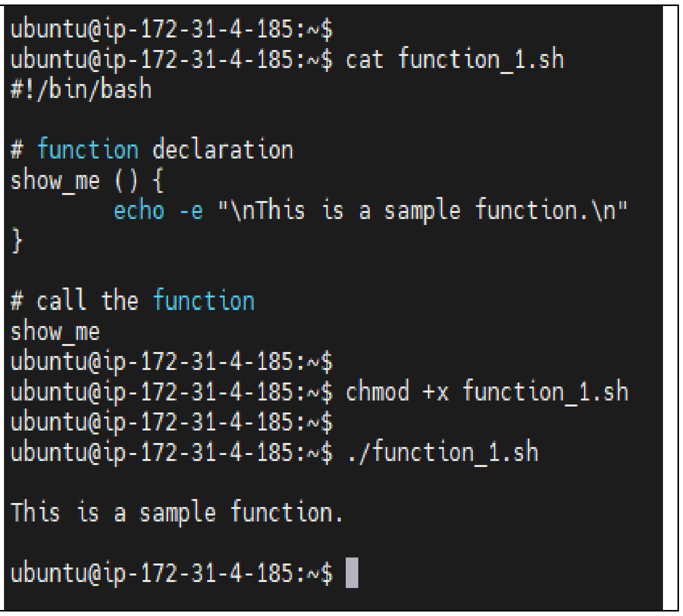
A function is a block of code that performs one, or more, operations. Functions are useful for executing procedures multiple times in a shell script. To use a function in a script, it must first be declared, then invoked.

For example, in this script, named **function_1.sh** I have declared a function named **show_me** which displays a simple message (with an empty line before and after) onscreen.

```
#!/bin/bash
# function declaration
show_me () {
   echo -e "\nThis is a sample function.\n"
}
# call the function
show_me
```

# make executable

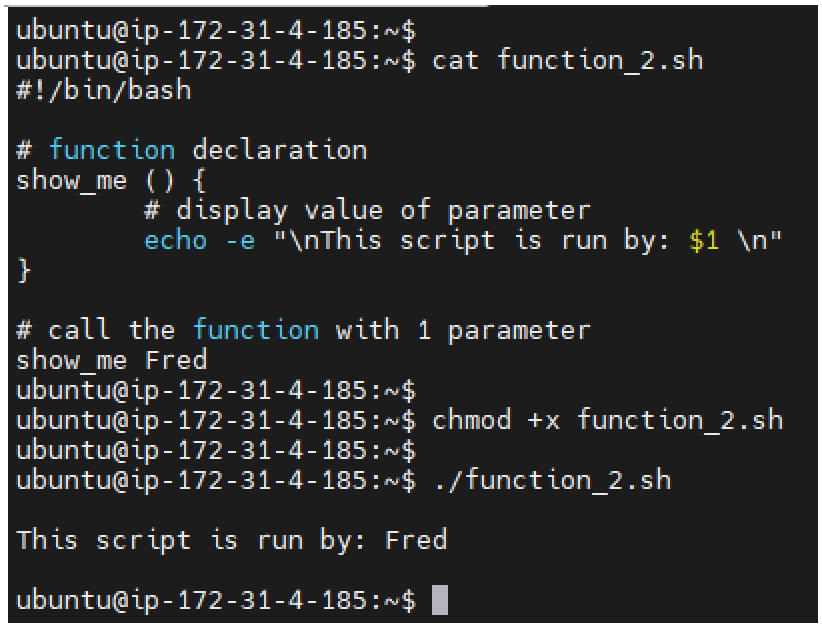
# execute script



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat function_1.sh
#!/bin/bash

# function declaration
show_me () {
        echo -e "\nThis is a sample function.\n"
}

# call the function
show_me
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ chmod +x function_1.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./function_1.sh

This is a sample function.

ubuntu@ip-172-31-4-185:~$
```

In my next example, **function_2.sh**, I will demonstrate how functions can take parameters.

```
#!/bin/bash
# function declaration
show_me () {
        # display value of parameter
        echo -e "\nThis script is run by: $1 \n"
}
# call the function with 1 parameter
show_me Fred
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat function_2.sh
#!/bin/bash

# function declaration
show_me () {
        # display value of parameter
        echo -e "\nThis script is run by: $1 \n"
}

# call the function with 1 parameter
show_me Fred
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ chmod +x function_2.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./function_2.sh

This script is run by: Fred

ubuntu@ip-172-31-4-185:~$
```

Note also that I was able to reuse most of **function_1.sh** to speed up development time.

Tests Operators

bash provides a set of file test operators, that can be used with the if statement, including:

Condition	Meaning
-d file	True if file is a directory.
-e file	True if file exists.
-f file	True if file is a regular file.
-r file	True if file is readable by you.
-s file	True if file exists and is not empty.
-w file	True if file is writable by you.
-x file	True if file is executable by you.

bash also provides a number of string test operators, that can be used with the if statement:

-z STRING	True if string is empty.
-n STRING	True if string is not empty.
STRING1 = STRING2	True if the strings are equal.
STRING1 != STRING2	True if the strings are not equal.

Finally, **bash** provides a number of arithmetic test operators that can be used with the if statement:

arg1 –eq arg2	True if arg1 is equal to arg2.
arg1 –ne arg2	True if arg1 is not equal to arg2.
arg1 –lt arg2	True if arg1 is less than arg2.
arg1 –le arg2	True if arg1 is less than or equal to arg2.
arg1 –gt arg2	True if arg1 is greater than arg2.
arg1 –ge arg2	True if arg1 is greater than or equal to arg2.

if statements

The if statement is a construct that allows for conditional decision making. When an if statement is used, the resulting actions depend on the evaluation of conditions such as:

- Numerical or string comparisons
- Return value of a command (0 for success)
- File existence or permissions

There three possible definitions for the if statement as listed below:

if	if / else	if / elif / else

`if condition`
`then`
`        statements`
`fi`
	`if condition`
	`then`
	`        statements`
	`else`
	`        statements`
	`fi`
		`if condition`
		`then`
		`        statements`
		`elif condition`
		`then`
		`        statements`
		`else`
		`        statements`
		`fi`

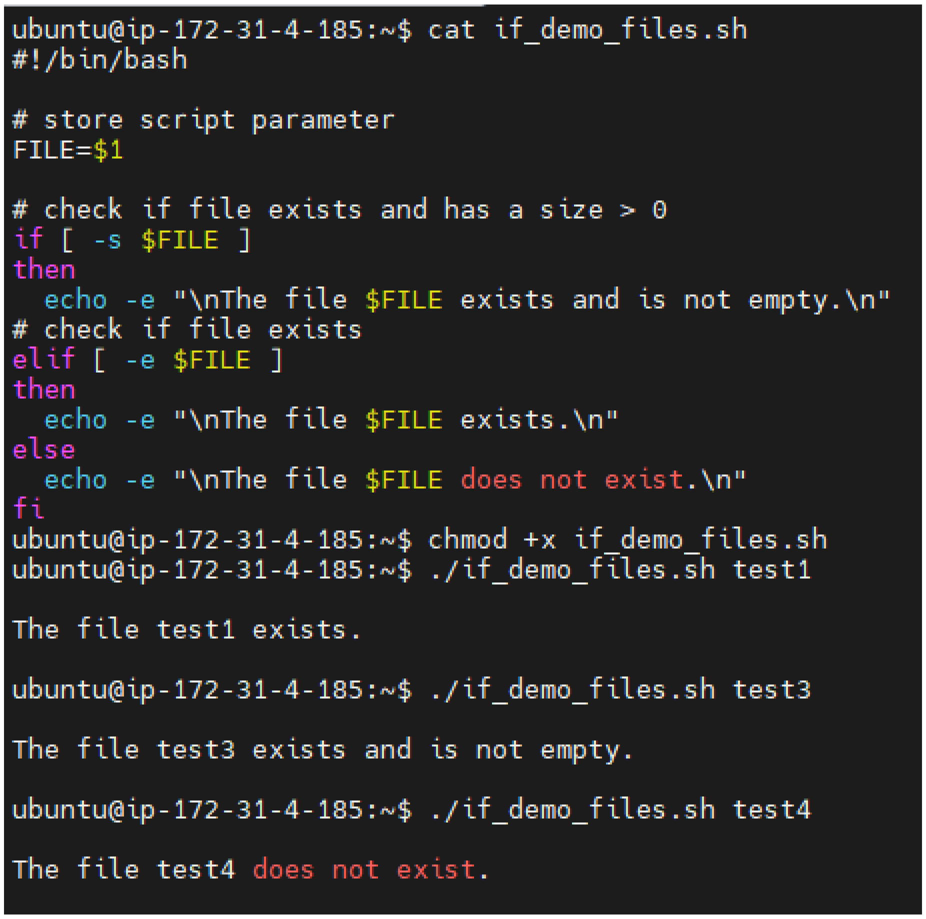
Before proceeding, we will create a few test files:

```
$ touch test1 test2 test3
$ echo "This is test3" > test3
```

Now, create a script, **if_demo_files.sh**, to check whether files exist and for files that are not empty.

Remember to make it executable and execute the script.

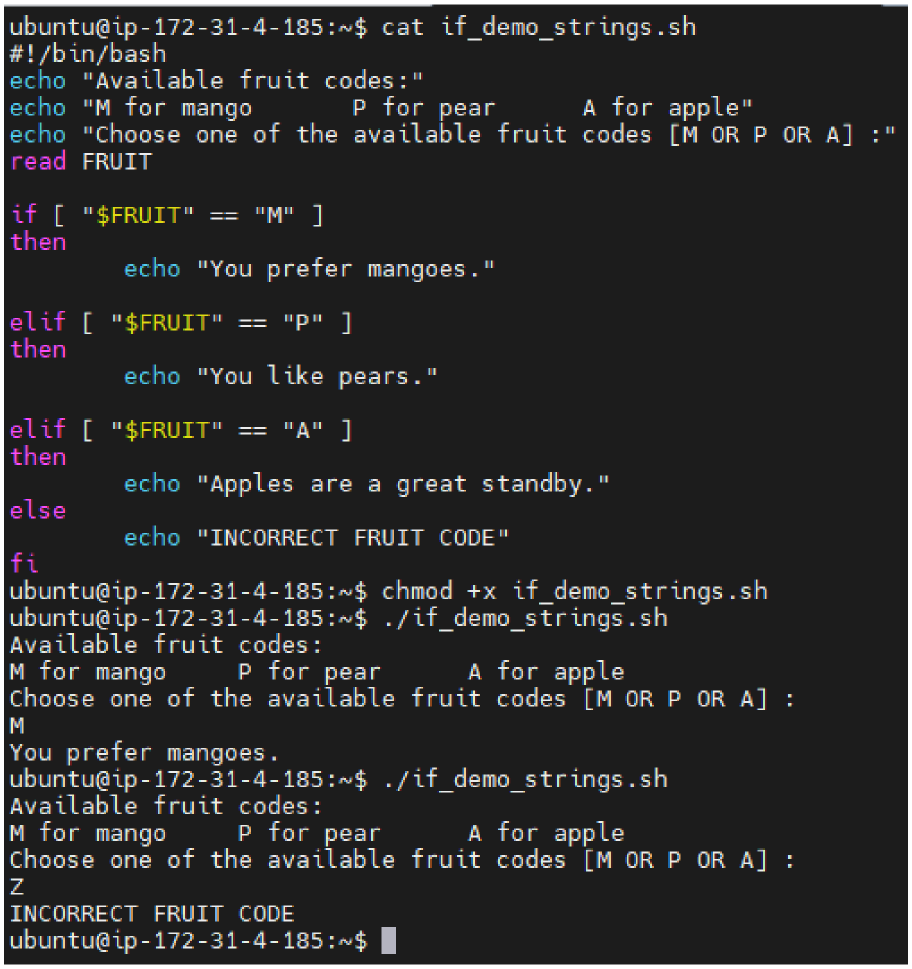
```
#!/bin/bash
# store script parameter
FILE=$1
# check if file exists and has a size > 0
if [ -s $FILE ]
then
  echo -e "\nThe file $FILE exists and is not empty.\n"
# check if file exists
elif [ -e $FILE ]
then
  echo -e "\nThe file $FILE exists.\n"
else
  echo -e "\nThe file $FILE does not exist.\n"
fi
```



```
ubuntu@ip-172-31-4-185:~$ cat if_demo_files.sh
#!/bin/bash

# store script parameter
FILE=$1

# check if file exists and has a size > 0
if [ -s $FILE ]
then
  echo -e "\nThe file $FILE exists and is not empty.\n"
# check if file exists
elif [ -e $FILE ]
then
  echo -e "\nThe file $FILE exists.\n"
else
  echo -e "\nThe file $FILE does not exist.\n"
fi
ubuntu@ip-172-31-4-185:~$ chmod +x if_demo_files.sh
ubuntu@ip-172-31-4-185:~$ ./if_demo_files.sh test1

The file test1 exists.

ubuntu@ip-172-31-4-185:~$ ./if_demo_files.sh test3

The file test3 exists and is not empty.

ubuntu@ip-172-31-4-185:~$ ./if_demo_files.sh test4

The file test4 does not exist.
```

The next script, **if_demo_strings.sh**, will prompt the user to enter an available fruit code and then display a message about the fruit selected.

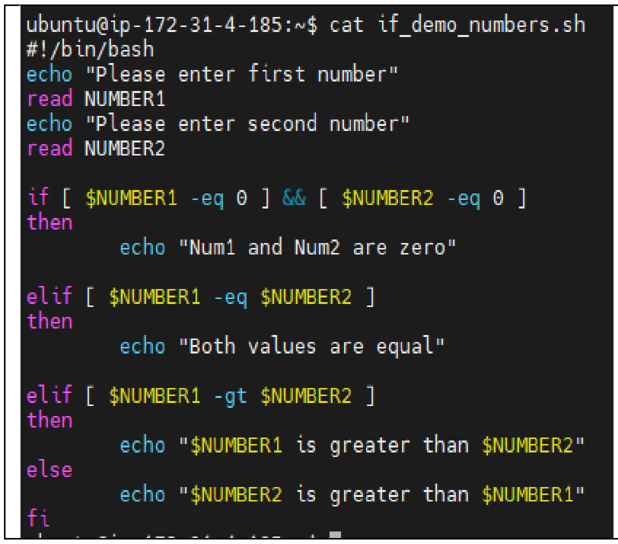
```
#!/bin/bash
echo "Available fruit codes:"
echo "M for mango   P for pear   A for apple"
echo "Choose one of the available fruit codes [M OR P OR A] :"
read FRUIT
if [ "$FRUIT" == "M" ]
then
      echo "You prefer mangoes."
elif [ "$FRUIT" == "P" ]
then
      echo "You like pears."
elif [ "$FRUIT" == "A" ]
then
      echo "Apples are a great standby."
else
      echo "INCORRECT FRUIT CODE"
fi
```



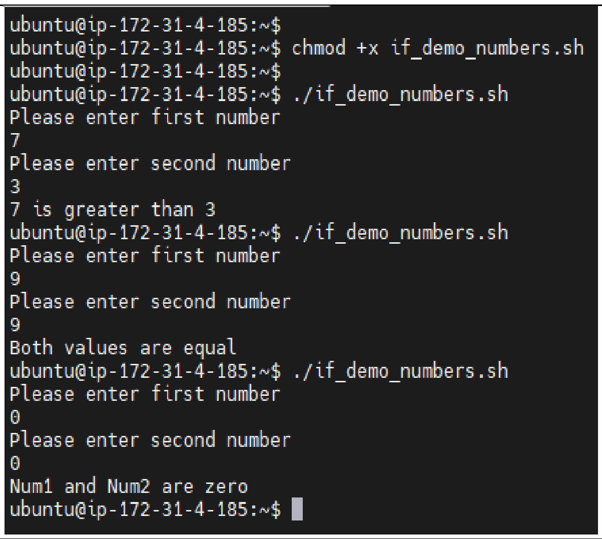
```
ubuntu@ip-172-31-4-185:~$ cat if_demo_strings.sh
#!/bin/bash
echo "Available fruit codes:"
echo "M for mango       P for pear      A for apple"
echo "Choose one of the available fruit codes [M OR P OR A] :"
read FRUIT

if [ "$FRUIT" == "M" ]
then
        echo "You prefer mangoes."

elif [ "$FRUIT" == "P" ]
then
        echo "You like pears."

elif [ "$FRUIT" == "A" ]
then
        echo "Apples are a great standby."
else
        echo "INCORRECT FRUIT CODE"
fi
ubuntu@ip-172-31-4-185:~$ chmod +x if_demo_strings.sh
ubuntu@ip-172-31-4-185:~$ ./if_demo_strings.sh
Available fruit codes:
M for mango     P for pear      A for apple
Choose one of the available fruit codes [M OR P OR A] :
M
You prefer mangoes.
ubuntu@ip-172-31-4-185:~$ ./if_demo_strings.sh
Available fruit codes:
M for mango     P for pear      A for apple
Choose one of the available fruit codes [M OR P OR A] :
Z
INCORRECT FRUIT CODE
ubuntu@ip-172-31-4-185:~$
```

We will next perform arithmetic tests using the if statement. Create a file named **if_demo_numbers.sh** and don't forget to make it executable before testing it.

```
#!/bin/bash
echo "Please enter first number"
read NUMBER1
echo "Please enter second number"
read NUMBER2

if [ $NUMBER1 -eq 0 ] && [ $NUMBER2 -eq 0 ]
then
        echo "Num1 and Num2 are zero"
elif [ $NUMBER1 -eq $NUMBER2 ]
then
        echo "Both values are equal"
elif [ $NUMBER1 -gt $NUMBER2 ]
then
        echo "$NUMBER1 is greater than $NUMBER2"
else
        echo "$NUMBER2 is greater than $NUMBER1"
fi
```



```
ubuntu@ip-172-31-4-185:~$ cat if_demo_numbers.sh
#!/bin/bash
echo "Please enter first number"
read NUMBER1
echo "Please enter second number"
read NUMBER2

if [ $NUMBER1 -eq 0 ] && [ $NUMBER2 -eq 0 ]
then
        echo "Num1 and Num2 are zero"

elif [ $NUMBER1 -eq $NUMBER2 ]
then
        echo "Both values are equal"

elif [ $NUMBER1 -gt $NUMBER2 ]
then
        echo "$NUMBER1 is greater than $NUMBER2"
else
        echo "$NUMBER2 is greater than $NUMBER1"
fi
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ chmod +x if_demo_numbers.sh
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ ./if_demo_numbers.sh
Please enter first number
7
Please enter second number
3
7 is greater than 3
ubuntu@ip-172-31-4-185:~$ ./if_demo_numbers.sh
Please enter first number
9
Please enter second number
9
Both values are equal
ubuntu@ip-172-31-4-185:~$ ./if_demo_numbers.sh
Please enter first number
0
Please enter second number
0
Num1 and Num2 are zero
ubuntu@ip-172-31-4-185:~$
```

case statements

The case statement is used in scenarios where the actual value of a variable can lead to different execution paths. case statements are often used to handle command-line options.

Below are some of the advantages of using the case statement:

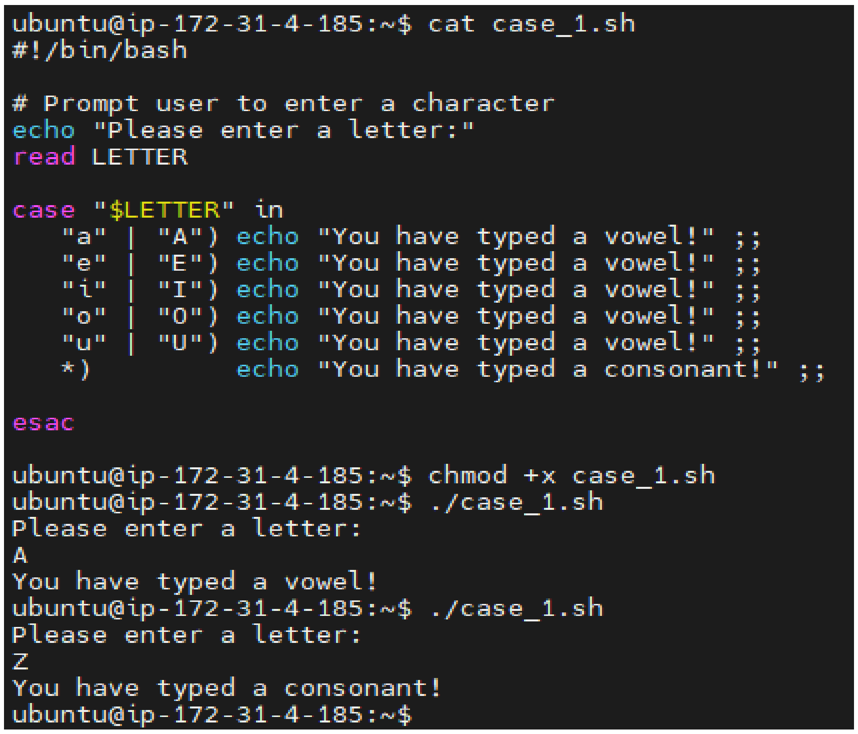
- It is easier to read and write.
- It is a good alternative to nested, multi-level if-then-else-fi code blocks.
- It enables you to compare a variable against several values at once.
- It reduces the complexity of a program.

Here is the basic structure of the case statement:

```
case expression in
    pattern1) execute commands;;
    pattern2) execute commands;;
    pattern3) execute commands;;
    pattern4) execute commands;;
    * )       execute some default commands or nothing ;;
esac
```

In the following script, **case_1.sh**, I am checking to see whether a vowel or consonant was entered.

```
#!/bin/bash
echo "Please enter a letter:"
read LETTER
case "$LETTER" in
   "a" | "A") echo "You have typed a vowel!" ;;
   "e" | "E") echo "You have typed a vowel!" ;;
   "i" | "I") echo "You have typed a vowel!" ;;
   "o" | "O") echo "You have typed a vowel!" ;;
   "u" | "U") echo "You have typed a vowel!" ;;
   *)         echo "You have typed a consonant!" ;;
esac
```



```
ubuntu@ip-172-31-4-185:~$ cat case_1.sh
#!/bin/bash

# Prompt user to enter a character
echo "Please enter a letter:"
read LETTER

case "$LETTER" in
   "a" | "A") echo "You have typed a vowel!" ;;
   "e" | "E") echo "You have typed a vowel!" ;;
   "i" | "I") echo "You have typed a vowel!" ;;
   "o" | "O") echo "You have typed a vowel!" ;;
   "u" | "U") echo "You have typed a vowel!" ;;
   *)         echo "You have typed a consonant!" ;;

esac

ubuntu@ip-172-31-4-185:~$ chmod +x case_1.sh
ubuntu@ip-172-31-4-185:~$ ./case_1.sh
Please enter a letter:
A
You have typed a vowel!
ubuntu@ip-172-31-4-185:~$ ./case_1.sh
Please enter a letter:
Z
You have typed a consonant!
ubuntu@ip-172-31-4-185:~$
```

String Fundamentals

To save the length of a variable use the following syntax:

`length1=${#string1}`

At times, you may not need an entire string. Bash provides a way to extract a substring from a string.

The following syntax is used to extract from **string** all characters starting from **position**.

`${string:position}`

We can also extract a substring (**length**) from **string** starting from **position**.

`${string:position:length}`

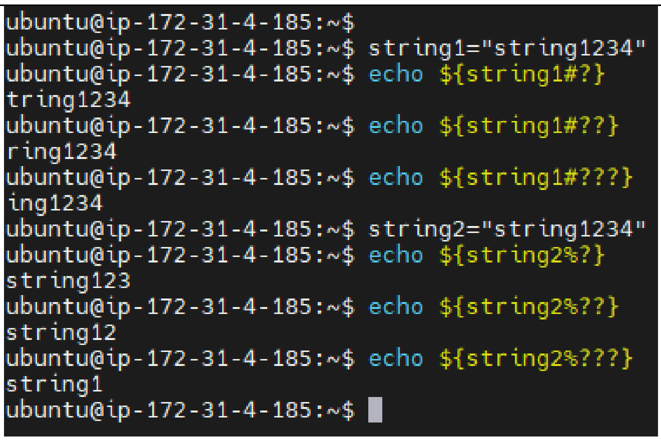
To extract the first character of a string we can specify:

`${string:0:1}`

Here 0 is the character to begin the extraction from and 1 is the number of characters to be extracted.

We can use pattern matching to remove characters from either end of the string.

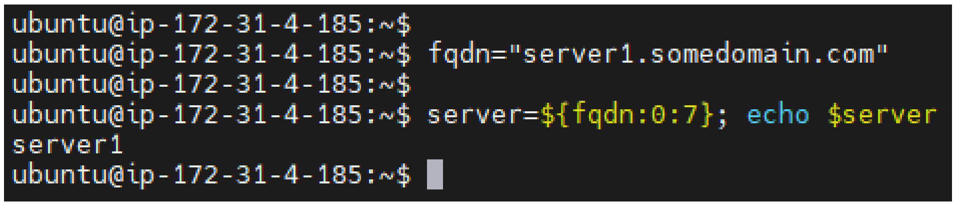
```
# remove characters from beginning
# of string
$ string1="string1234"
$ echo {string1#?}
$ echo {string1#??}
$ echo {string1#???}
# remove characters from end of string
$ string2="1234string"
$ echo {name2%?}
$ echo {name2%??}
$ echo {name2%???}
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ string1="string1234"
ubuntu@ip-172-31-4-185:~$ echo ${string1#?}
tring1234
ubuntu@ip-172-31-4-185:~$ echo ${string1#??}
ring1234
ubuntu@ip-172-31-4-185:~$ echo ${string1#???}
ing1234
ubuntu@ip-172-31-4-185:~$ string2="string1234"
ubuntu@ip-172-31-4-185:~$ echo ${string2%?}
string123
ubuntu@ip-172-31-4-185:~$ echo ${string2%??}
string12
ubuntu@ip-172-31-4-185:~$ echo ${string2%???}
string1
ubuntu@ip-172-31-4-185:~$
```

We can extract the server name from the fully qualified domain name using substring notation.

```
$ fqdn="server1.somedomain.com"
$ server=${fqdn:0:7}; echo $server     # extract 7 chars starting from position 0
```

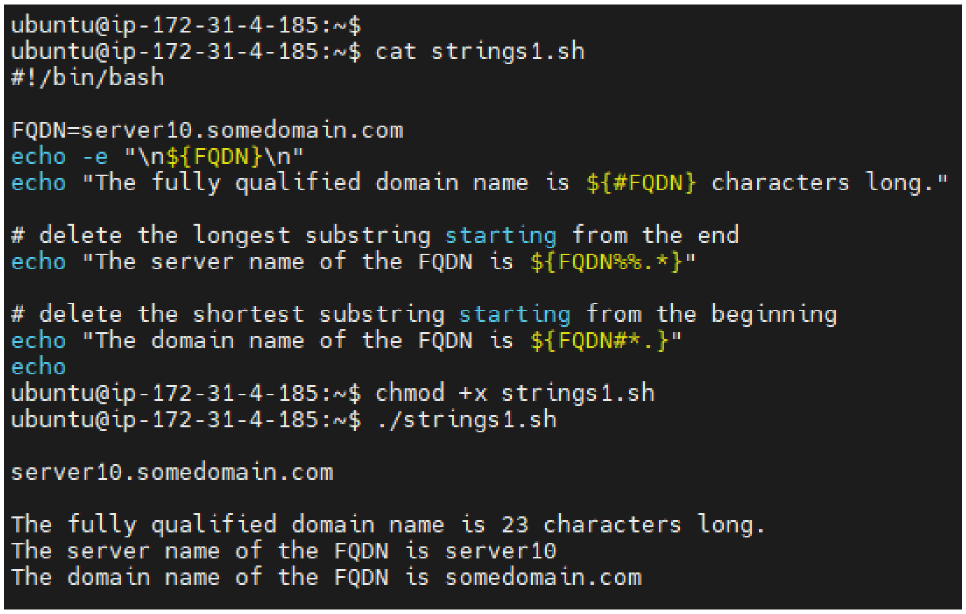


```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ fqdn="server1.somedomain.com"
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ server=${fqdn:0:7}; echo $server
server1
ubuntu@ip-172-31-4-185:~$
```

Note that If the server was named **server10**, the previous example would not work, resulting in **server1**.

We can use pattern matching to determine the server and domain names in the FQDN.

```
#!/bin/bash
FQDN=server10.somedomain.com
echo "The fully qualified domain name is ${#name} characters long."

# delete the longest substring starting from the end
echo "The server name of the FQDN is ${FQDN%%.*}

# delete the shortest substring starting from the beginning
echo "The domain name of the FQDN is ${FQDN#*.}
```

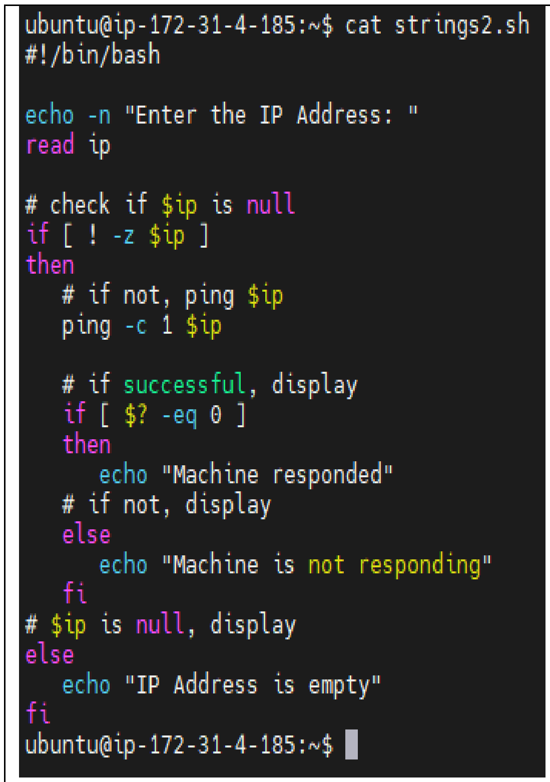


```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat strings1.sh
#!/bin/bash

FQDN=server10.somedomain.com
echo -e "\n${FQDN}\n"
echo "The fully qualified domain name is ${#FQDN} characters long."

# delete the longest substring starting from the end
echo "The server name of the FQDN is ${FQDN%%.*}"

# delete the shortest substring starting from the beginning
echo "The domain name of the FQDN is ${FQDN#*.}"
echo
ubuntu@ip-172-31-4-185:~$ chmod +x strings1.sh
ubuntu@ip-172-31-4-185:~$ ./strings1.sh

server10.somedomain.com

The fully qualified domain name is 23 characters long.
The server name of the FQDN is server10
The domain name of the FQDN is somedomain.com
```

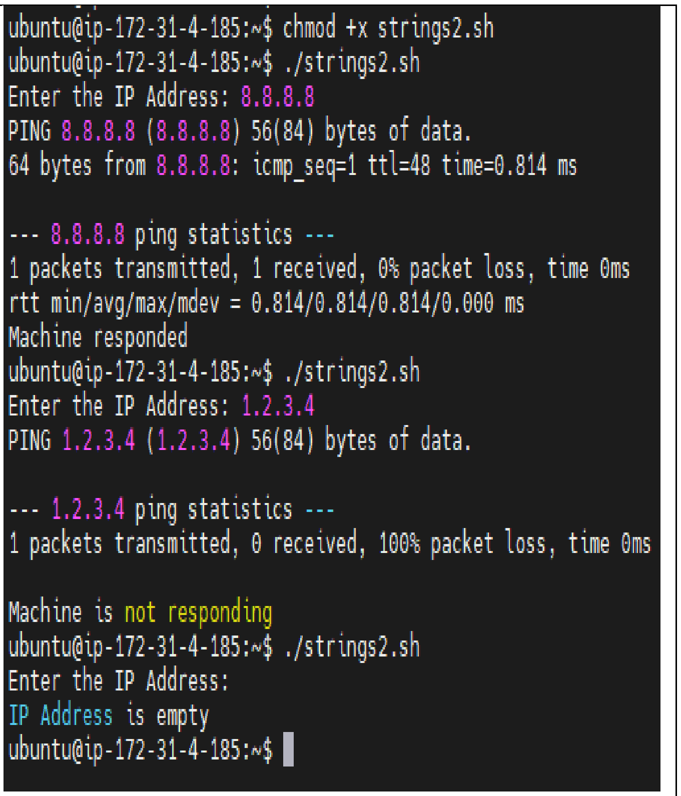
Testing for Strings

We will now prompt the user for an IP address and test to see if it is reachable using the **ping** command.

```
#!/bin/bash
echo -n "Enter the IP Address: "
read ip

# check if $ip is null
if [ ! -z $ip ]
then
   # if not, ping $ip
   ping -c 1 $ip
   # if successful, display
   if [ $? -eq 0 ]
   then
      echo "Machine responded"
   # if not, display
   else
      echo "Machine is not responding"
   fi
# $ip is null, display
else
   echo "IP Address is empty"
fi
```



```
ubuntu@ip-172-31-4-185:~$ cat strings2.sh
#!/bin/bash

echo -n "Enter the IP Address: "
read ip

# check if $ip is null
if [ ! -z $ip ]
then
   # if not, ping $ip
   ping -c 1 $ip

   # if successful, display
   if [ $? -eq 0 ]
   then
      echo "Machine responded"
   # if not, display
   else
      echo "Machine is not responding"
   fi
# $ip is null, display
else
   echo "IP Address is empty"
fi
ubuntu@ip-172-31-4-185:~$
```



```
ubuntu@ip-172-31-4-185:~$ chmod +x strings2.sh
ubuntu@ip-172-31-4-185:~$ ./strings2.sh
Enter the IP Address: 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=0.814 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.814/0.814/0.814/0.000 ms
Machine responded
ubuntu@ip-172-31-4-185:~$ ./strings2.sh
Enter the IP Address: 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.

--- 1.2.3.4 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

Machine is not responding
ubuntu@ip-172-31-4-185:~$ ./strings2.sh
Enter the IP Address:
IP Address is empty
ubuntu@ip-172-31-4-185:~$
```

Looping Constructs

Three types of loops are available to us: **for**, **while** & **until**

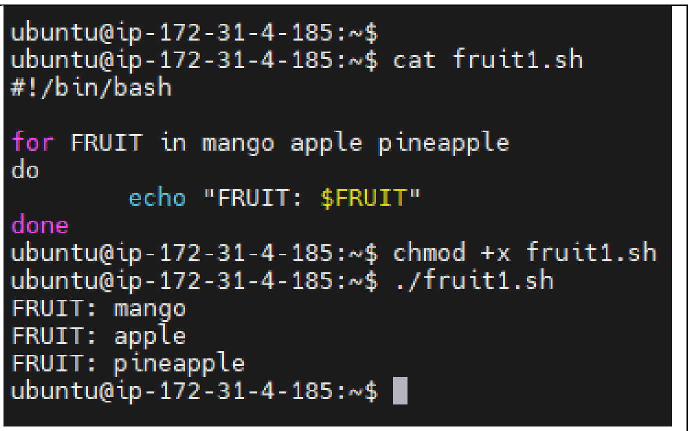
All these loops are used for repeating a set of statements until the exit condition is true.

The for loop operates on each element of a list of items. The syntax for the for loop is:

```
for VARIABLE_NAME in LIST
do
        command 1
        command 2
        command N
done
```

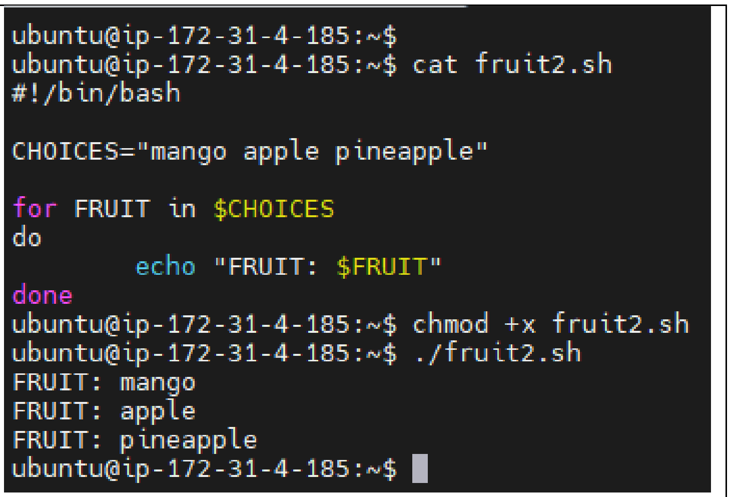
Here are a few examples using for loops.

```
#!/bin/bash
for FRUIT in mango apple pineapple
do
    echo "FRUIT: $FRUIT"
done
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat fruit1.sh
#!/bin/bash

for FRUIT in mango apple pineapple
do
        echo "FRUIT: $FRUIT"
done
ubuntu@ip-172-31-4-185:~$ chmod +x fruit1.sh
ubuntu@ip-172-31-4-185:~$ ./fruit1.sh
FRUIT: mango
FRUIT: apple
FRUIT: pineapple
ubuntu@ip-172-31-4-185:~$
```

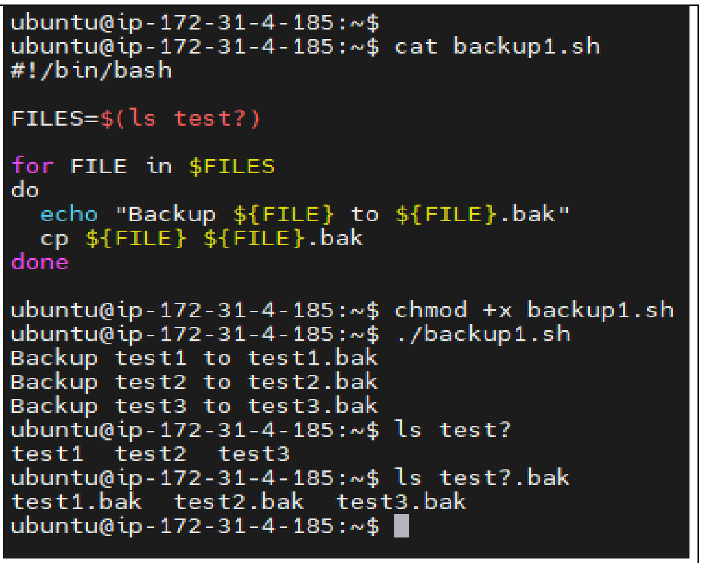
```
#!/bin/bash
CHOICES="mango apple pineapple"
for FRUIT in $CHOICES
do
    echo "FRUIT: $FRUIT"
done
```



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat fruit2.sh
#!/bin/bash

CHOICES="mango apple pineapple"

for FRUIT in $CHOICES
do
        echo "FRUIT: $FRUIT"
done
ubuntu@ip-172-31-4-185:~$ chmod +x fruit2.sh
ubuntu@ip-172-31-4-185:~$ ./fruit2.sh
FRUIT: mango
FRUIT: apple
FRUIT: pineapple
ubuntu@ip-172-31-4-185:~$
```

We will now use a for loop to make backup copies of the test files we created earlier: **test1**, **test2**, **test3**

```
#!/bin/bash
FILES=$(ls test?)
for FILE in FILES
do
  echo "Backup ${FILE} to ${FILE}.bak"
  cp ${FILE} ${FILE}.bak
done
$ chmod +x backup1.sh
$ ./backup1.sh
$ ls test?
$ ls test?.bak
```



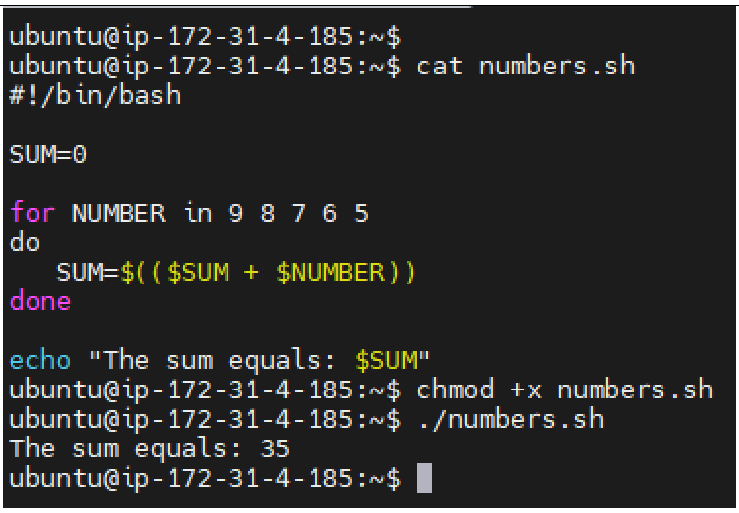
```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat backup1.sh
#!/bin/bash

FILES=$(ls test?)

for FILE in $FILES
do
  echo "Backup ${FILE} to ${FILE}.bak"
  cp ${FILE} ${FILE}.bak
done

ubuntu@ip-172-31-4-185:~$ chmod +x backup1.sh
ubuntu@ip-172-31-4-185:~$ ./backup1.sh
Backup test1 to test1.bak
Backup test2 to test2.bak
Backup test3 to test3.bak
ubuntu@ip-172-31-4-185:~$ ls test?
test1  test2  test3
ubuntu@ip-172-31-4-185:~$ ls test?.bak
test1.bak  test2.bak  test3.bak
ubuntu@ip-172-31-4-185:~$
```

We can also use an arithmetic expression to calculate the sum of a list of numbers

```
#!/bin/bash
SUM=0
for NUMBER in 9 8 7 6 5
do
   SUM=$(($SUM + $NUMBER))
done
echo "The sum equals: $SUM"
```

# make executable

# execute script



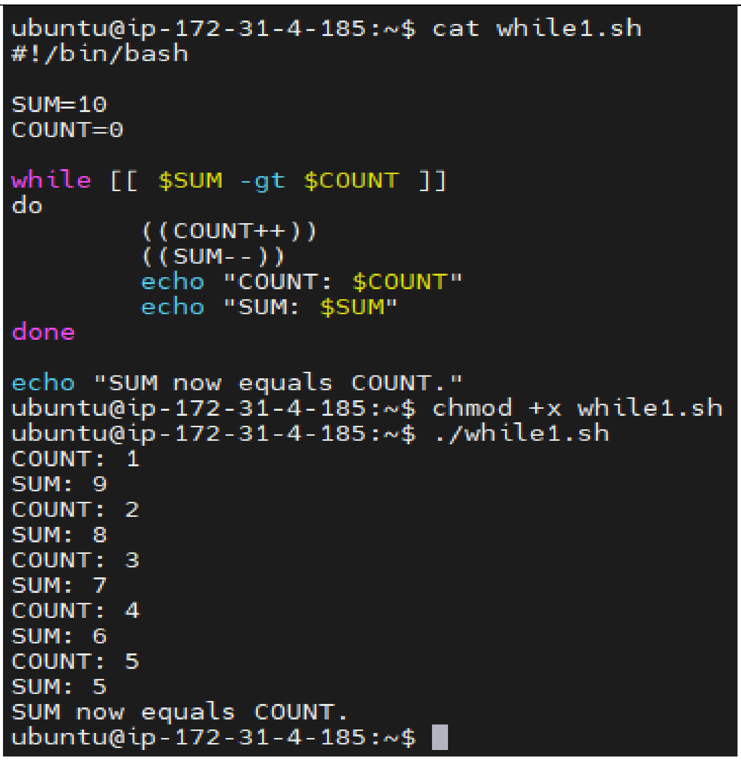
```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat numbers.sh
#!/bin/bash

SUM=0

for NUMBER in 9 8 7 6 5
do
   SUM=$(($SUM + $NUMBER))
done

echo "The sum equals: $SUM"
ubuntu@ip-172-31-4-185:~$ chmod +x numbers.sh
ubuntu@ip-172-31-4-185:~$ ./numbers.sh
The sum equals: 35
ubuntu@ip-172-31-4-185:~$
```

The while loop repeats a set of statements as long as the control command returns true. The syntax is:

```
while CONDITION_IS_TRUE
do
        command 1
        command 2
        command N
done
```

The set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition. Often it is enclosed within square brackets.

```
#!/bin/bash
SUM=10
COUNT=0
while [[ $SUM -gt $COUNT ]]
do
    (($COUNT++))
    (($SUM--))
    echo "COUNT: $COUNT"
    echo "SUM: $SUM"
done
echo "SUM now equals COUNT."
```



```
ubuntu@ip-172-31-4-185:~$ cat while1.sh
#!/bin/bash

SUM=10
COUNT=0

while [[ $SUM -gt $COUNT ]]
do
        ((COUNT++))
        ((SUM--))
        echo "COUNT: $COUNT"
        echo "SUM: $SUM"
done

echo "SUM now equals COUNT."
ubuntu@ip-172-31-4-185:~$ chmod +x while1.sh
ubuntu@ip-172-31-4-185:~$ ./while1.sh
COUNT: 1
SUM: 9
COUNT: 2
SUM: 8
COUNT: 3
SUM: 7
COUNT: 4
SUM: 6
COUNT: 5
SUM: 5
SUM now equals COUNT.
ubuntu@ip-172-31-4-185:~$
```

For the next demonstration, create an input file named **file1** with the following:

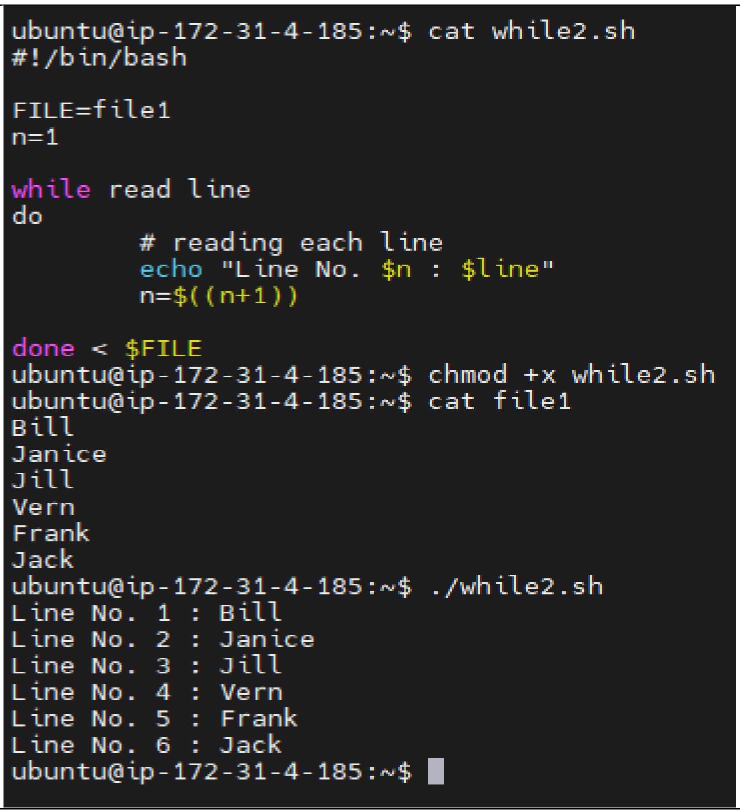
```
$ echo -e "Bill\nJanice\nJill\nVern\nFrank\nJack" > file1
```

Now, we will read file1 line by line using a while loop and display each line preceded by it's line number.

```
#
#!/bin/bash
$FILE=file1
n=1
while read line
do
   # reading each line
   echo "Line No. $n : $line"
   n=$((n+1))
done < $FILE
```

# make executable
# check file1 contents

# execute script



```
ubuntu@ip-172-31-4-185:~$ cat while2.sh
#!/bin/bash

FILE=file1
n=1

while read line
do
        # reading each line
        echo "Line No. $n : $line"
        n=$((n+1))

done < $FILE
ubuntu@ip-172-31-4-185:~$ chmod +x while2.sh
ubuntu@ip-172-31-4-185:~$ cat file1
Bill
Janice
Jill
Vern
Frank
Jack
ubuntu@ip-172-31-4-185:~$ ./while2.sh
Line No. 1 : Bill
Line No. 2 : Janice
Line No. 3 : Jill
Line No. 4 : Vern
Line No. 5 : Frank
Line No. 6 : Jack
ubuntu@ip-172-31-4-185:~$
```

The until loop repeats a set of statements as long as the control command is false. Thus it is essentially the opposite of the while loop. The syntax is:

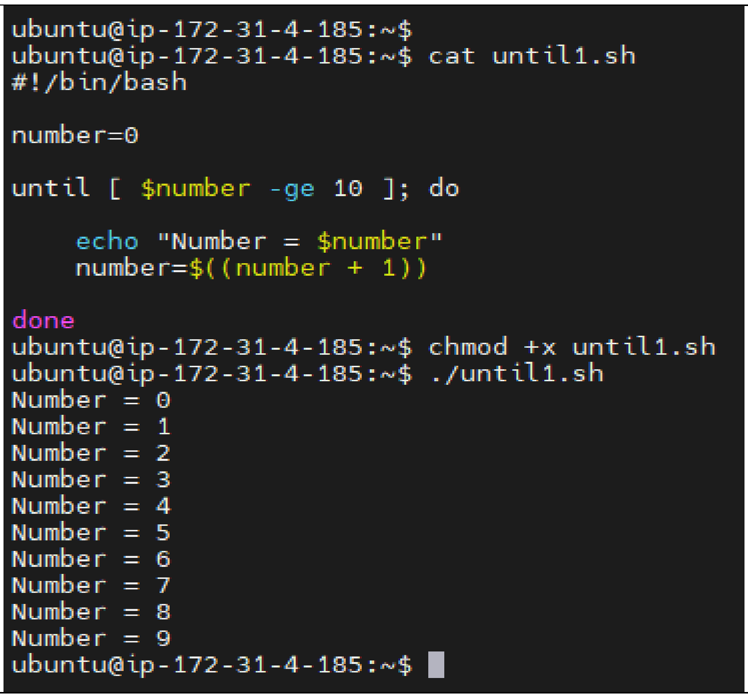
```
until CONDITION_IS_FALSE
do
        command 1
        command 2
        command N
done
```

Similar to the while loop, the set of commands that need to be repeated should be enclosed between do and done. You can use any command or operator as the condition.

```
#!/bin/bash

number=0
until [ $number -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

# make executable

# execute script



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ cat until1.sh
#!/bin/bash

number=0

until [ $number -ge 10 ]; do

    echo "Number = $number"
    number=$((number + 1))

done
ubuntu@ip-172-31-4-185:~$ chmod +x until1.sh
ubuntu@ip-172-31-4-185:~$ ./until1.sh
Number = 0
Number = 1
Number = 2
Number = 3
Number = 4
Number = 5
Number = 6
Number = 7
Number = 8
Number = 9
ubuntu@ip-172-31-4-185:~$
```

Script Debugging

Before fixing an error (or bug), it is vital to know its source.

In bash shell scripting, you can run a script in debug mode by doing

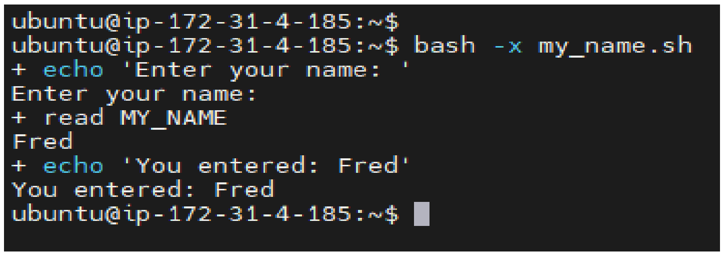
```
$ bash –x ./script_file
```

Debug mode helps identify the error because:

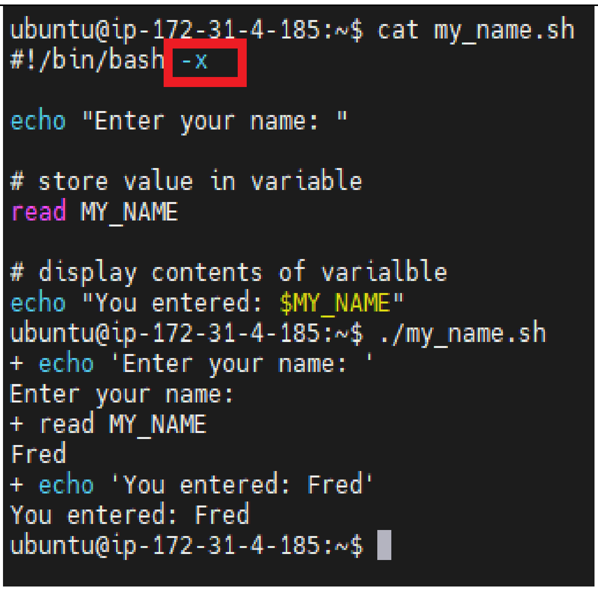
- It traces and prefixes each command with the + character.
- It displays each command before executing it.
- it can debug the entire script by adding **-x** after the shell declaration #!/bin/bash
- It can debug only selected parts of a script (if desired) with:

```
set -x    # turns on debugging
...
set +x    # turns off debugging
```

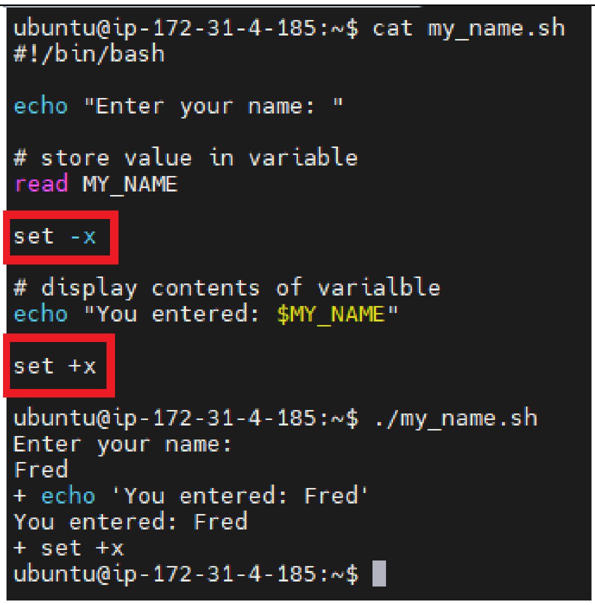
Here I am demonstrating the three methods of debugging a script.



```
ubuntu@ip-172-31-4-185:~$
ubuntu@ip-172-31-4-185:~$ bash -x my_name.sh
+ echo 'Enter your name: '
Enter your name:
+ read MY_NAME
Fred
+ echo 'You entered: Fred'
You entered: Fred
ubuntu@ip-172-31-4-185:~$
```



```
ubuntu@ip-172-31-4-185:~$ cat my_name.sh
#!/bin/bash -x

echo "Enter your name: "

# store value in variable
read MY_NAME

# display contents of varialble
echo "You entered: $MY_NAME"
ubuntu@ip-172-31-4-185:~$ ./my_name.sh
+ echo 'Enter your name: '
Enter your name:
+ read MY_NAME
Fred
+ echo 'You entered: Fred'
You entered: Fred
ubuntu@ip-172-31-4-185:~$
```



```
ubuntu@ip-172-31-4-185:~$ cat my_name.sh
#!/bin/bash

echo "Enter your name: "

# store value in variable
read MY_NAME

set -x

# display contents of varialble
echo "You entered: $MY_NAME"

set +x

ubuntu@ip-172-31-4-185:~$ ./my_name.sh
Enter your name:
Fred
+ echo 'You entered: Fred'
You entered: Fred
+ set +x
ubuntu@ip-172-31-4-185:~$
```

I hope you have enjoyed completing this tutorial and found it helpful.

I have a few other shell scripts. One used to create Linux user accounts. Another that checks the status of services and restarts them if necessary. These services (**Nginx, PostgreSQL** & **Gunicorn**) support a number of my secured Django / Flask apps running on my Digital Ocean droplet. I also have a script that monitors my OCI (Oracle Cloud Infrastructure) deployment of the same secured Django / Flask websites, along with the required services (listed above), as dockerized containers. Feel free to browse through them under the **Scripting** tab **here**.

My main tutorials page can be accessed **here**.

Back to Top