Contents

[Shell Scripting on Windows 10 3](#_Toc94530622)

[The Hash-Bang Hack 4](#_Toc94530623)

[New Line = New Command 4](#_Toc94530624)

[Comment Often with #s 4](#_Toc94530625)

[Commands Are Surrounded By Parentheses 5](#_Toc94530626)

[Make the Script Executable and then Run It 5](#_Toc94530627)

[How to Work With Windows Files in a Bash Script 6](#_Toc94530628)

[How to Incorporate Bash Commands into a Batch or PowerShell Script 6](#_Toc94530629)

[Your First Script 7](#_Toc94530630)

[Types of Shells 10](#_Toc94530631)

[Command-Line Shell 10](#_Toc94530632)

[Graphical Shell 11](#_Toc94530633)

[What is BASH Shell? 11](#_Toc94530634)

[What is BASH Scripting? 11](#_Toc94530635)

[Advantages 11](#_Toc94530636)

[Disadvantages 11](#_Toc94530637)

[Using Echo Command 11](#_Toc94530638)

[Using Comments 12](#_Toc94530639)

[Using Variables 13](#_Toc94530640)

[Getting User Input 15](#_Toc94530641)

[Using Command Arguments 15](#_Toc94530642)

[Using Loops 16](#_Toc94530643)

[While Loop 16](#_Toc94530644)

[For Loop 17](#_Toc94530645)

[Using Conditional Statements 17](#_Toc94530646)

[if statement 17](#_Toc94530647)

[if else statement 18](#_Toc94530648)

[if elif statement 18](#_Toc94530649)

[Using Functions 19](#_Toc94530650)

[Display String Length 20](#_Toc94530651)

[Concatenating Strings 20](#_Toc94530652)

[Extracting String 21](#_Toc94530653)

[Find and Replace String 21](#_Toc94530654)

[Check Even/Odd Number 22](#_Toc94530655)

[Generate Factorial of Number 23](#_Toc94530656)

[Creating Directories 23](#_Toc94530657)

[Reading Files 24](#_Toc94530658)

[Deleting Files 25](#_Toc94530659)

[Print Files With Line Count 25](#_Toc94530660)

[Print Number of Files and Folders 26](#_Toc94530661)

[Error handling in Bash scripts 26](#_Toc94530662)

[Exit status 27](#_Toc94530663)

[Best practices 27](#_Toc94530664)

[Standard error redirection 28](#_Toc94530665)

[Error Handling 28](#_Toc94530666)

[Exit codes 28](#_Toc94530667)

[Exit on error 28](#_Toc94530668)

# Shell Scripting on Windows 10

When writing shell scripts on Windows, bear in mind that Windows and UNIX-like systems like Linux use different “end of line” characters in text files in shell scripts.

In other words, this means that you can’t simply write a shell script in Notepad. Save the file in Notepad and it won’t be interpreted properly by Bash. However, you can use more advanced text editors–for example, Notepad++ allows you to give a file UNIX end-of-line characters by clicking Edit > EOL Conversion > UNIX/OSX Format.

Graphical user interface, text, application, email

Description automatically generated

However, you’re better off just writing the shell script in the Bash environment itself. The Ubuntu-based Bash environment comes with both the vi and nano text editors. The vi editor is more powerful, but if you’ve never used it before, you may want to start with nano. It’s easier to use if you’re new.

For example, to create a bash script in nano, you’d run the following command in bash:

nano ~/myscript.sh

This would open the Nano text editor pointed at a file named “myscript.sh” in your user account’s home directory. (The “~” character represents your home directory, so the full path is /home/username/myscript.sh.)

Rectangle

Description automatically generated with low confidence

Start your shell script with the line:

#!/bin/bash

Enter the commands you want to run, each one on its own line. The script will run each command in turn. Add a “#” character before a line to treat it as a “comment”, something which helps you and other people understand the script, but which isn’t run as a command. The same techniques will work in Bash on Ubuntu on Windows.

Note that there’s no way to run Windows programs from within the Bash environment. You’re restricted to Linux terminal commands and utilities, just as you would be on a typical Linux system.

For example, let’s just use a basic “hello world” script as an example here:

#!/bin/bash

# set the STRING variable

STRING="Hello World!"

# print the contents of the variable on screen

echo $STRING

If you’re using the Nano text editor, you can save the file by pressing Ctrl+O and then Enter. Close the editor by pressing Ctrl+X.

As I mentioned before, every script file is essentially plain text. That doesn’t mean you can write what you want all willy-nilly, though. When a text file is attempted to be executed, shells will parse through them for clues as to whether they’re scripts or not, and how to handle everything properly. Because of this, there are a few guidelines you need to know.

1. Every script should being with “#!/bin/bash”
2. Every new line is a new command
3. Comment lines start with a #
4. Commands are surrounded by ()

## The Hash-Bang Hack

When a shell parses through a text file, the most direct way to identify the file as a script is by making your first line:

#!/bin/bash

If you use another shell, substitute its path here. Comment lines start with hashes (#), but adding the bang (!) and the shell path after it is a sort of hack that will bypass this comment rule and will force the script to execute with the shell that this line points to.

## **New Line = New Command**

Every new line should be considered a new command, or a component of a larger system. If/then/else statements, for example, will take over multiple lines, but each component of that system is in a new line. Don’t let a command bleed over into the next line, as this can truncate the previous command and give you an error on the next line. If your text editor is doing that, you should turn off text-wrapping to be on the safe side. You can turn off text wrapping in nano bit hitting ALT+L.

## **Comment Often with #s**

If you start a line with a #, the line is ignored. This turns it into a comment line, where you can remind yourself of what the output of the previous command was, or what the next command will do. Again, turn off text wrapping, or break you comment into multiple lines that all begin with a hash. Using lots of comments is a good practice to keep, as it lets you and other people tweak your scripts more easily. The only exception is the aforementioned Hash-Bang hack, so don’t follow #s with !s. ;-)

## **Commands Are Surrounded By Parentheses**

In older days, command substitutions were done with single tick marks (`, shares the ~ key). We’re not going to be touching on this yet, but as most people go off and explore after learning the basics, it’s probably a good idea to mention that you should use parentheses instead. This is mainly because when you nest — put commands inside other commands — parentheses work better.

Text

Description automatically generated

## Make the Script Executable and then Run It

You’ll probably want the make the script executable so you can run it more easily. On Linux, that means you need to give the script file the executable permission. To do so, run the following command in the terminal, pointing it at your script:

chmod +x ~/myscript.sh

To run the script, you can now just run it in the terminal by typing its path. Whenever you want to launch the script in the future, just open the Bash shell and type the path to the script.

~/myscript.sh

(If the script is in the current directory, you can run it with ./myscript.sh)

Text

Description automatically generated with medium confidence

## How to Work With Windows Files in a Bash Script

To access Windows files in the script, you’ll need to specify their path under /mnt/c, not their Windows path. For example, if you wanted to specify the C:\Users\Bob\Downloads\test.txt file, you’d need to specify the /mnt/c/Users/Bob/Downloads/test.txt path

## How to Incorporate Bash Commands into a Batch or PowerShell Script

Lastly, if you have an existing batch file or PowerShell script you want to incorporate commands into, you can run Bash commands directly using the bash -c command.

For example, to run a Linux command in a Command Prompt or PowerShell window, you can run the following command:

bash -c "command"

This trick allows you to add Bash commands into batch files or PowerShell scripts. The Bash shell window will appear when a Bash command is running.

**Update**: If you have multiple Linux environments installed, you can use the wslconfig command to choose the default Linux environment used when you run the bash -c command.

Text

Description automatically generated

To create a shortcut to a Bash script from within Windows, just create a shortcut like normal. For the shortcut’s target, use the bash -c command we outlined above and point it at the Bash script you created.

For example, you’d point a shortcut at ” bash -c "~/myscript.sh" ” to run the example script above. You can also just run this command from a Command Prompt or PowerShell window, too.

## Your First Script

Let’s start with a simple script that allows you to copy files and append dates to the end of the filename. Let’s call it “datecp”. First, let’s check to see if that name conflicts with something:

which datecp

You can see that there’s no output of the which command, so we’re all set to use this name.

Let’s create a blank file in the ~/bin folder:

touch ~/bin/datecp

And, let’s change the permission now, before we forget:

chmod u+x ~/bin/datecp

Let’s start building our script then. Open up that file in your text editor of choice. Like I said, I like the simplicity of nano.

nano ~/bin/datecp

And, let’s go ahead and put in the prerequisite first line, and a comment about what this script does.

#!/bin/bash

@ This will copy a file, appending the date and time

# to the end of the file name.

Next, let’s declare a variable. If you’ve ever taken algebra, you probably know what a that is. A variable allows us to store information and do things with it. Variables can “expand” when referenced elsewhere. That is, instead of displaying their name, they will display their stored contents. You can later tell that same variable to store different information, and any instruction that occurs after that will use the new information. It’s a really fancy placeholder.

What will we put in out variable? Well, let’s store the date and time! To do this, we’ll call upon the date command.

Look at the screenshot below for how to build the output of the date command:

~$ date +%D

01/31/22

~$ date +%m

01

~$ date +%m\_%d\_%Y

01\_31\_2022

~$ date +%m\_%d\_%y

01\_31\_22

~$ date +%m\_%d\_%y-%l.%M.%S

01\_31\_22-12.40.23

~$ date +%m\_%d\_%y-%H.%M.%S

01\_31\_22-12.41.36

~$

You can see that by adding different variables that start with %, you can change the output of the command to what you want. For more information, you can look at the manual page for the date command.

Let’s use that last iteration of the date command, “date +%m\_%d\_%y-%H.%M.%S” and use that in our script ~/bin/datecp.

#!/bin/bash

@ This will copy a file, appending the date and time

# to the end of the file name.

date +%m\_%d\_%y-%H.%M.%S

If we were to save this script right now, we could run it and it would give us the output of the date command like we’d expect:

~$ datecp

01\_31\_22-12.41.36

You can also run the script from Bash as;

~$ sh /bin/datecp

But, let’s do something different. Let’s give a variable name, like date\_formatted to this command. The proper syntax for this is as follows:

variable=$(command –options arguments)

And for us, we’d build it like this:

date\_formatted=$(date +%m\_%d\_%y-%H.%M.%S)

#!/bin/bash

@ This will copy a file, appending the date and time

# to the end of the file name.

date\_formatted=$(date +%m\_%d\_%y-%H.%M.%S)

echo “This is the Date and Time: “ $date\_formatted

This is what we call command substitution. We’re essentially telling bash that whenever the variable “date\_formatted” shows up, to run the command inside the parentheses. Then, whatever output the commands gives should be displayed instead of the name of the variable, “date\_formatted”.

Here’s an example script and its output:

~$ datecp

This is the Date and Time: 01\_31\_22-12.41.36

Note that there are two spaces in the output. The space within the quotes of the echo command and the space in front of the variable are both displayed. Don’t use spaces if you don’t want them to show up. Also note that without this added “echo” line, the script would give absolutely no output.

Let’s get back to our script. Let’s next add in the copying part of the command.

cp –iv $1 $2.$date\_formatted

#!/bin/bash

@ This will copy a file, appending the date and time

# to the end of the file name.

date\_formatted=$(date +%m\_%d\_%y-%H.%M.%S)

cp -iv $1 $2.$date\_formatted

This will invoke the copy command, with the –i and –v options. The former (“interactive”) will ask you for verification before overwriting a file, and the latter (“verbose”) will display on the command line what is being done.

Next, you can see I’ve added the “$1” option. When scripting, a dollar sign ($) followed by a number will denote that numbered argument of the script when it was invoked. For example, in the following command:

cp –iv Trogdor2.mp3 ringtone.mp3

The first argument is “Trogdor2.mp3” and the second argument is “ringtone.mp3”.

Looking back at our script, we can see that we’re referencing two arguments:

cp -iv $1 $2.$date\_formatted

This means that when we run the script, we’ll need to provide two arguments for the script to run correctly. The first argument, $1, is the file that will be copied, and is substituted as the “cp –iv” command’s first argument.

The second argument, $2, will act as the output file for the same command. But, you can also see that it’s different. We’ve added a period and we’ve referenced the “date\_formatted” variable from above. Curious as to what this does?

Here’s what happens when the script is run:

~$ datecp somefile.txt /temp/somefile\_copied.txt

‘somefile.txt’ -> ‘/temp/somefile\_copied.txt.01\_31\_22-12.41.36

You can see that the output file is listed as whatever I entered for $2, followed by a period, then the output of the date command! Makes sense, right?

Now when I run the datecp command, it will run this script and allow me to copy any file to a new location, and automatically add the date and time to end of the filename. Useful for archiving stuff!

## Types of Shells

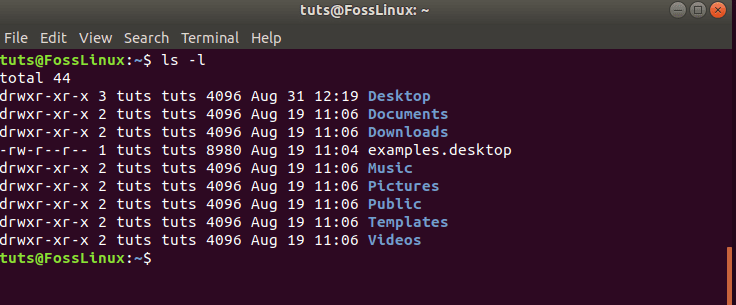
Shell is a unique program that provides the user an interface to interact with kernel accepting human-readable commands and then converts it to kernel understandable language. Shell, in a Linux operating system, can take input from the user in the form of commands, processes it, and then displays an output. You can access Shell using Terminal in Linux.

Shell has two categories:

1. Command-Line Shell
2. Graphical Shell

### Command-Line Shell

A shell can be accessed by a user using command-line interfaces. We have programs like the terminal in (Linux or Mac) and Command Prompt in Windows to get input in the form of human-readable commands and then display output in the same command-line interface.



### Graphical Shell

Graphical shell provides users a Graphical User Interface (GUI) to interact, perform operations like opening, closing, saving files. Windows OS and Ubuntu are great examples of GUI Shell (Desktop), where a user does not have to type commands for every operation. Still, behind every action, there is a shell command that executes to perform these actions.

### What is BASH Shell?

BASH (Bourne Again Shell) is the default command-line interpreter for most of the Linux Distros these days. It is an updated version of the earlier Bourne shell. If you are a Linux system administrator or a power user, you must have excellent knowledge of BASH shell commands to perform day to day tasks.

### What is BASH Scripting?

Mostly we use shell commands one by one in the terminal for our everyday tasks. Still, sometimes you have to perform complex tasks or repetitive tasks, which involves a series of commands being executed in a proper sequence. A shell can also take commands as input from a file, so to make our job easy, we can write these commands in a file and can execute them in the shell to avoid manual work. These files are called shell scripts.

### Advantages

The bash script has many advantages:

* It automates repetitive work that saves a lot of effort and time.
* You can create your power tool or utility.
* Bash scripts are portable; you may use a script on other Linux systems without any modification.
* It has the same set of the syntax that we use in standard terminal, so do not involves additional learning.
* You can quickly write a bash script with little help.
* It can provide interactive debugging while running tasks that help in case of error or issue.

### Disadvantages

The bash script can have disadvantages:

* Prone to errors, a single mistake can change the program’s flow and can be harmful.
* Slow execution speed.
* Have very minimal data structures, unlike other programming languages.
* Not well suited for large and complex tasks.

## Using Echo Command

Echo command is the most common and frequently used command in Linux. It is used to print text or output in the Bash. It has many options that perform different operations.

Syntax:

echo [options] [ARGUMENTS]

Here options are:

-n is used to suppress trailing new line

-e is used to interpret backslash-escaped characters

-E is used to disables the interpretation of the escape characters, and it is the default option for the echo command.

Create a new file echo.sh and add the below lines in it.

#!/bin/bash

echo "Foss Linux"

echo -n "I am a Linux User"

echo -e "\nGood \t Bye \t All"

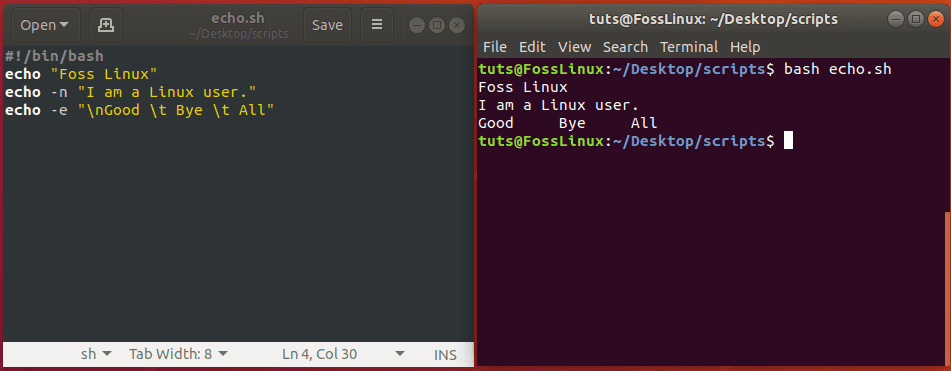
Here \n is an escape character for a new line, and \t is an escape character for the horizontal tab.

Make it an executable and then run the script as:

~$ chmod a+x echo.sh

~$ ./echo.sh

Output:



## Using Comments

Comments are a programmer’s remarks about the purpose of the code or logic. It’s a widespread practice to add comments so that in the future, anyone can understand code by just reading comments. Comments are part of code but ignored by the compiler. In the bash script, any line that starts with # is considered a comment. For example:

#!/bin/bash

# this is a comment

echo "Comment Example"

Here’ # this is a comment’ is a comment, and when we run this script compiler will ignore the line.

Comments can be:

1. Single Line Comment
2. Multiple Line Comment

We use ‘#’ for single line comment and: ‘content’ for multiple line comments. Check the below command for both single and numerous comments in a bash script named comments.sh.

#!/bin/bash

: '

This script calculates

sum of 2 and 8.

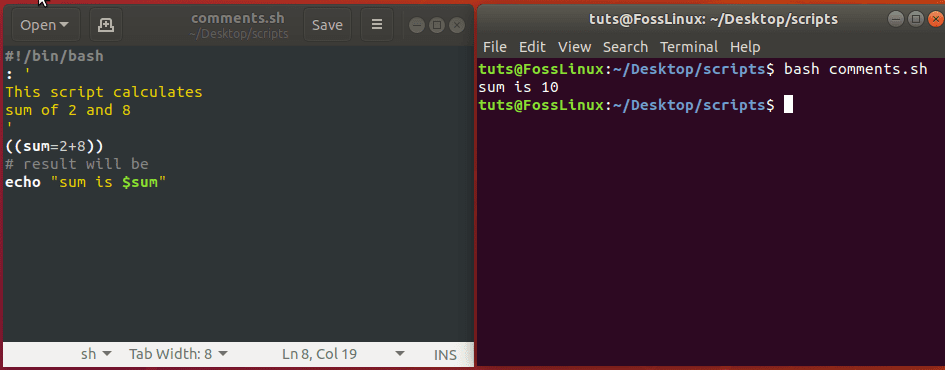
'

((sum=2+8))

# result will be

echo "sum is $sum"

Output:



## Using Variables

Variables are named symbols used to store values temporarily. It can be a string or numeric value that we may use at any place within the script. You can make variables and assign them values. Variable names should be descriptive so that you can understand the purpose you created that variable.

We have three kinds of variables in bash scripts:

1. Special Variables:

The following are the other unique preset variables:

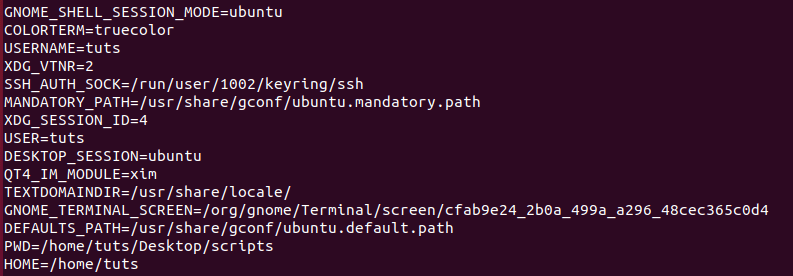
* $#: number of command line parameters that were passed to the script.
* $@: All the parameters sent to the script.
* $?: The end status of the last process to execute.
* $$: The Process ID of the current script.
* $USER: The user executing the script.
* $HOSTNAME: The hostname of the machine executing the script.
* $SECONDS: The number of seconds the script has been running for.
* $RANDOM: Returns a random number.
* $LINENO: Returns the current line number of the script.

1. Environment Variables:

To see the active environment variables in your Bash session, use the command:

env | less

Output:

environment variables

1. User-Defined Variables:

User-defined variables are those which are set by us in our script. For example, we have variable ‘year’ to store the current year like below.

year=2020

And we can later use

echo $year

you can see that we used $  to reference its value.

So now create a file variables.sh and add the below lines in it.

#!/bin/bash

website=www.fosslinux.com

year=2020

# Getting user name from special variables

name=$USER

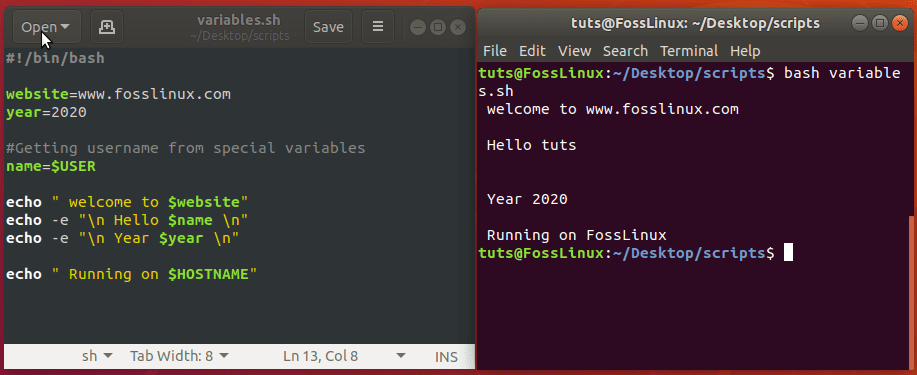
echo "Welcome to $website"

echo -e "Hello $name \n"

echo -e "Year = $year \n"

echo "Running on $HOSTNAME"

Output:



## Getting User Input

Getting user input is very crucial for making a script interactive, so for this purpose in bash script, we use ‘read’ command. File: userinput.sh:

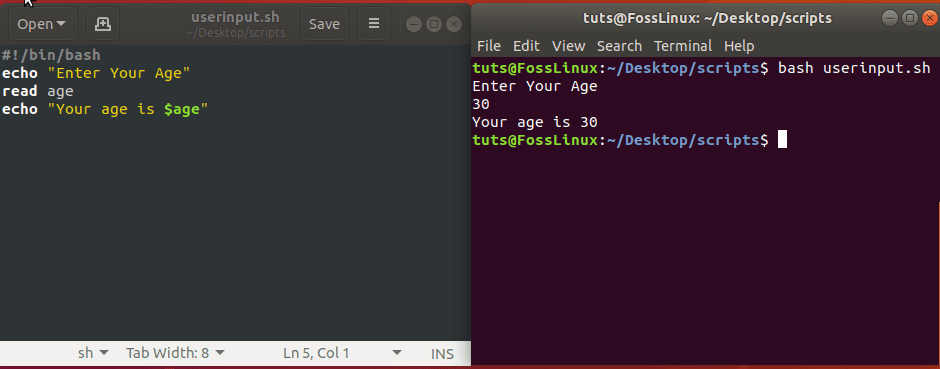
#!/bin/bash

echo "Enter Your Age"

read age

echo "Your age is $age"

Output:



## Using Command Arguments

We can also read user input from command arguments, just like any other programming language. We can then use these arguments in our scripts as $1, $2, and so on, depending on the number of arguments we have provided. Create a file’ cmdargs.sh’ and copy the below lines in it.

#!/bin/bash

echo "Total arguments : $#"

echo "Username: $1"

echo "Age: $2"

echo "Full Name: $3"

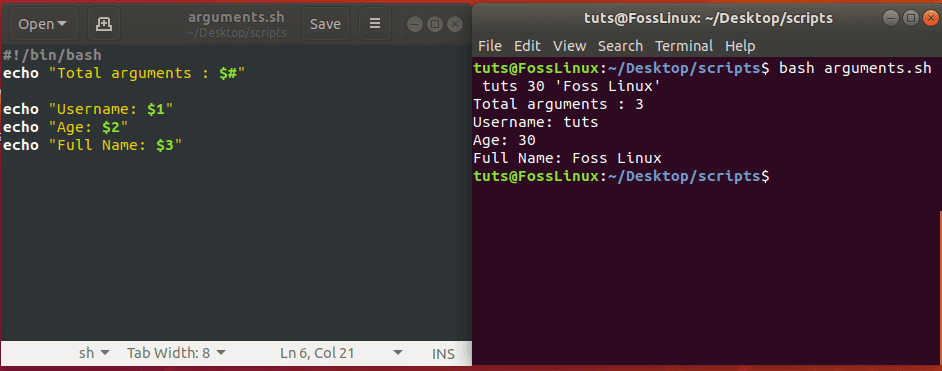
Now run ‘arguments.sh’ script file with three additional parameters after its name.

$ bash cmdargs.sh tuts 30 'Foss Linux'

OR from BASH:

$ ./cmdargs.sh ajay 30 'Ajay Singala'

Output:



## Using Loops

Loops are used in every programming language where you need to execute the same code repetitively. There are two types of loops in bash script while and for loops. We will see each one by one.

### While Loop

While it is used when you need to repeat the line of code an unknown number of times until it satisfies certain conditions. Here is how it is formed:

#!/bin/bash

while [CONDITION]

do

[COMMANDS]

done

The condition is evaluated before executing the commands at every iteration, and it will keep executing until the condition evaluates to false, and the loop will be terminated.

#!/bin/bash

i=0

while [ $i -le 4 ]

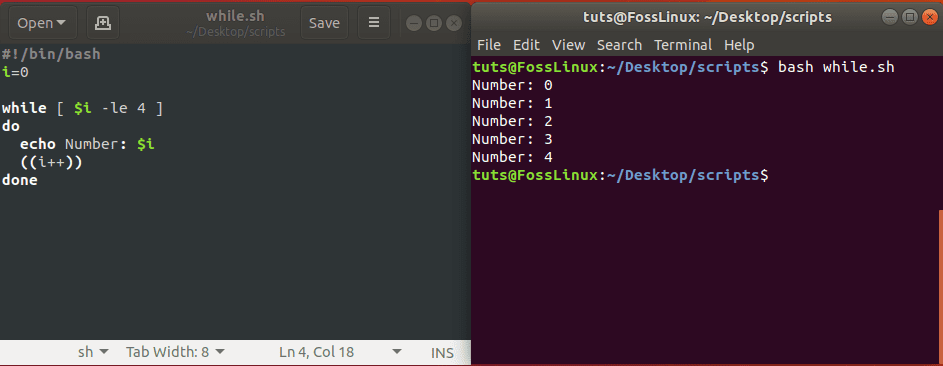
do

echo Number: $i

((i++))

done

Output:



### For Loop

The for loop iterates over a list of items and performs the given set of commands. The Bash for loop takes the following form:

#!/bin/bash

for item in [LIST]

do

[COMMANDS]

done

In the example below, the loop will iterate over each item and will generate a table of variable i.

#!/bin/bash

i=2

for (( counter=1; counter<=10; counter++ ))

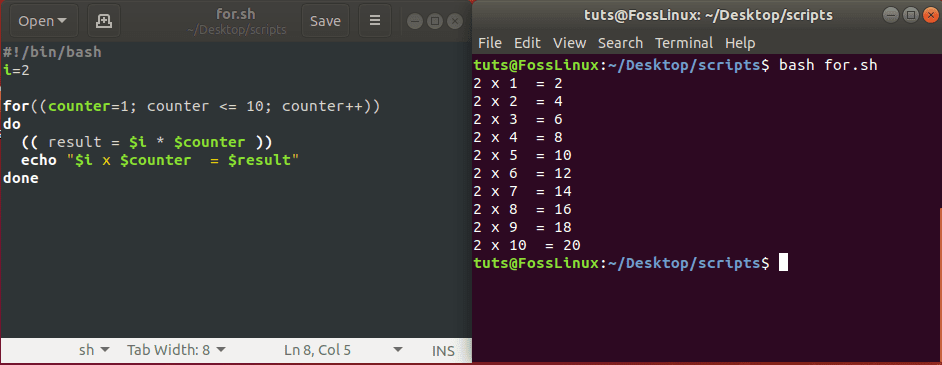
do

((result= $i \* $counter))

echo "$i x $counter = $result"

done

Output:



## Using Conditional Statements

Conditional statements are one of the fundamental concepts of any programming language. You make decisions based on certain conditions fulfilled. In the bash script, we have conditional blocks.

### if statement

In a bash script, if the condition has several forms but let’s look at the basic condition.

if Condition

then

STATEMENTS

fi

You can see if statements start with evaluating the condition and Run statements between ‘then’ and ‘fi’, provided the “If” condition evaluates to True otherwise statement will be ignored.

#!/bin/bash

echo -n "Enter a number: "

read VAR

if [[ $VAR -gt 10 ]]

then

echo "The variable is greater than 10."

fi

In the above example, the user will be asked to input a number, and if the number is more than 10, you will see output ‘The variable is greater than 10.’, otherwise you will not see anything.

### if else statement

Now we are going to add “if else” block as well, which will execute if the condition will be false.

if Condition

then

STATEMENTS1

else

STATEMENTS2

fi

So we will modify the above example.

#!/bin/bash

echo -n "Enter a number: "

read VAR

if [[ $VAR -gt 10 ]]

then

echo "The variable is greater than 10."

else

echo "The variable is equal or less than 10."

fi

If you execute the code and enter a number, the script will print a string based on whether the number is greater or less/equal to 10.

### if elif statement

Bash has an equivalent syntax for ‘else if’ as well.

if Condition1

then

STATEMENTS1

elif Condition2

then

STATEMENTS2

else

STATEMENTS3

fi

So after modifying the above example:

#!/bin/bash

echo -n "Enter a number: "

read VAR

if [[ $VAR -gt 10 ]]

then

echo "The variable is greater than 10."

elif [[ $VAR -eq 10 ]]

then

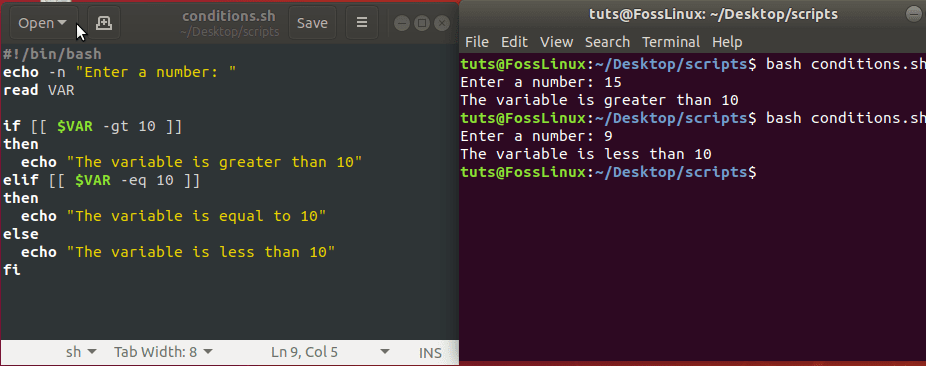
echo "The variable is equal to 10."

else

echo "The variable is less than 10."

fi

Output:



## Using Functions

Just like other programming languages, the bash script also has the concept of functions. It allows the user to write a custom code block that will be required to be reused again and again.

Syntax:

function FunctionName()

{

statements

}

Now we shall create a function ‘sum’ that will take input numbers from the user and will show the sum of these numbers as output.

#!/bin/bash

function Sum()

{

echo -n "Enter First Number: "

read a

echo -n "Enter Second Number: "

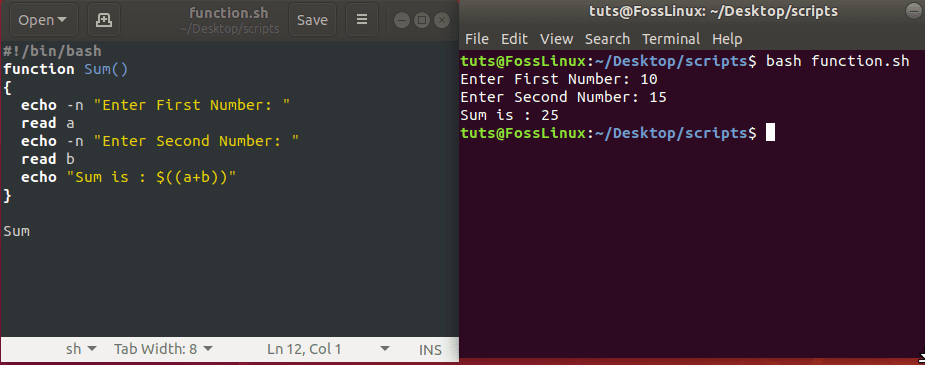
read b

echo "Sum is: $(( a+b ))"

}

Sum

Output:



## Display String Length

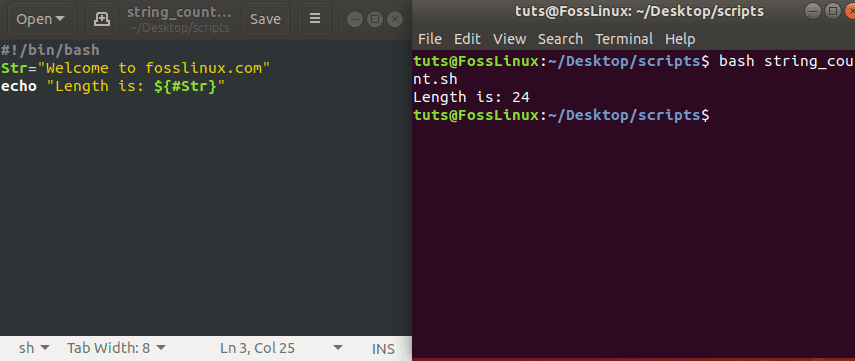
Processing strings is an essential part of bash scripting. Bash script has a straightforward way of getting a string variable’s length. In the example, we will show you how to get the length of a string in bash script.

#!/bin/bash

Str="Welcome to ajaysingala.com"

echo "Length is: ${#Str}"

Output:



## Concatenating Strings

Bash script provides an effortless way to handle string operations like the concatenation of multiple strings into a single string. In the example, we will show you how to do that.

#!/bin/bash

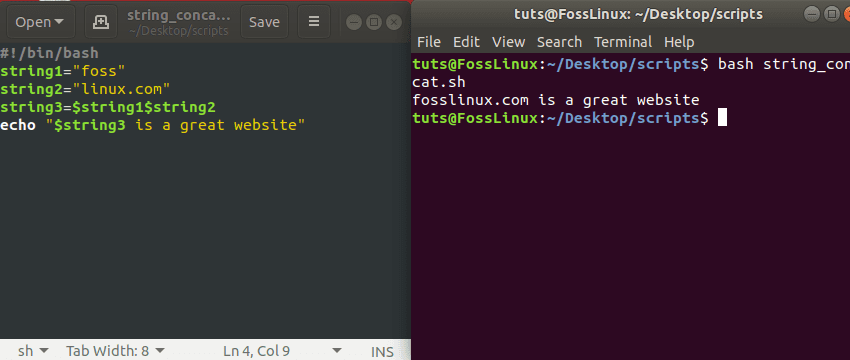
string1="ajay"

string2="singala.com"

string=$string1$string2

echo "$string is a great website."

Output:



## Extracting String

Bash gives a way to extract a substring from a string. The below example explains how to parse n characters starting from a particular position.

${string:position}

Extract substring from $string at $position

${string:position:length}

Extract $length of characters sub-string from $string starting from $position. In the below example, we will show you how it works.

#!/bin/bash

Str="Welcome to the ajaysingala.com"

# Extracting string from index 15

echo ${Str:15}

# Extracting string from index 15 of length 5

echo ${Str:15:5}

Output:

Graphical user interface, text

Description automatically generated

## Find and Replace String

Bash script has a handy and easy way to find and replace the text within a string. It can be used in two ways:

${string/pattern/replacement}

This will replace only the first match within the given string. To replace all matches, we will use it as shown below:

${string//pattern/replacement}

In another example, we will use both options to show you the difference in the output:

#! /bin/bash

Str="Path of the bash is /bin/bash"

# Finding and Replacing First match only

echo ${Str/bash/sh}

# Finding and Replacing all matches

echo ${Str//bash/sh}

Output:

Graphical user interface, text

Description automatically generated

## Check Even/Odd Number

In our next example, we will write a bash script that will accept an input number from the user and will display if a given number is an even number or odd number.

#!/bin/bash

echo -n "Enter The Number: "

read n

num=$(expr $n % 2)

if [ $num -eq 0 ]; then

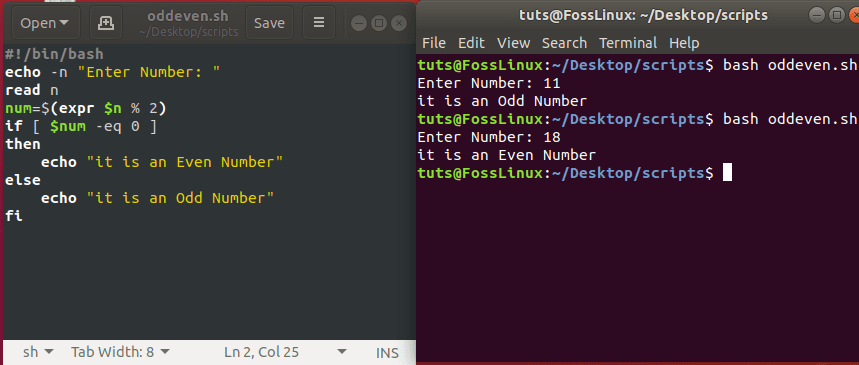
echo "It is a Even Number"

else

echo "It is a Odd Number"

fi

Output:



## Generate Factorial of Number

The following example will show you how to generate a factorial of a given number using a shell script.

#!/bin/bash

echo -n "Enter Number: "

read n

fact=1

while [ $n -gt 1 ]

do

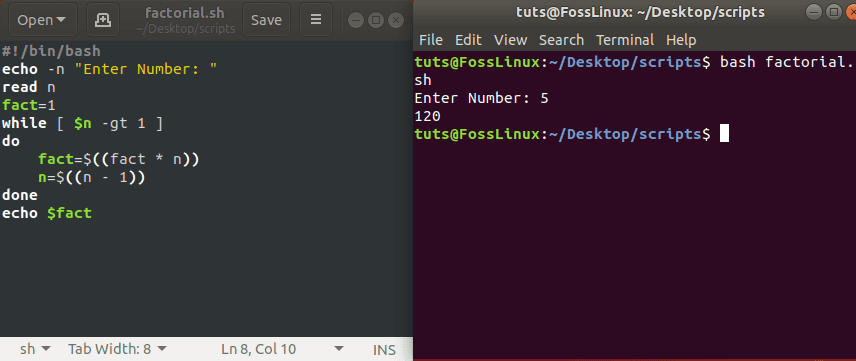
fact=$((fact \* n))

n=$((n - 1))

done

echo $fact

Output:



## Creating Directories

The following example will show you how to create a directory from within a shell script. The script will get the directory name from the user and will check if it already exists or not. In case it exists, you should see a message “Directory Exists”; otherwise, it will create a directory.

#!/bin/bash

echo -n "Enter directory name ->"

read dir

if [ -d "$dir" ]

then

echo "Directory exists"

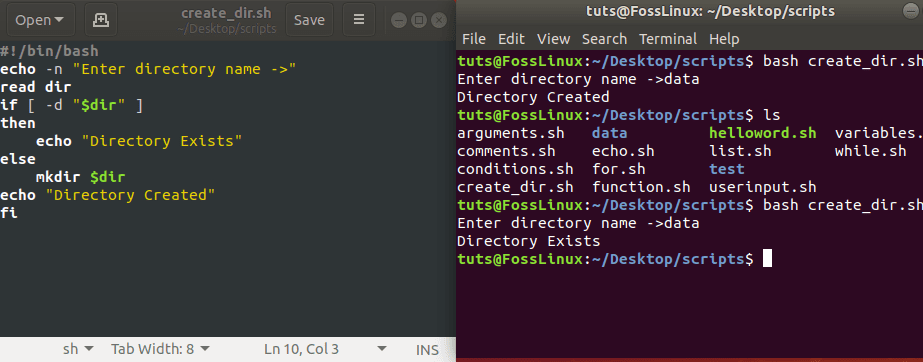
else

`mkdir $dir`

echo "Directory created"

fi

Output:



## Reading Files

Using Bash you can read files very effectively. The below example will showcase how to read a file using shell scripts. Create a file called ‘companies.txt’ with the following contents.

Google

Amazon

Microsoft

Macdonald

KFC

Apple

This script will read the above file and will display output.

#!/bin/bash

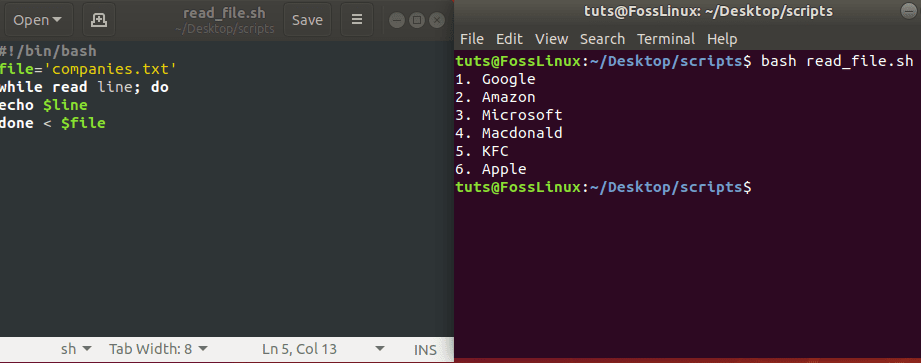
file='companies.txt'

while read line; do

echo $line

done < $file

Output:



## Deleting Files

Using a bash script, you can delete a file as well. In the example, the user will be asked to provide the filename as input and will delete it if it exists. It uses the Linux rm command for the deletion here.

#!/bin/bash

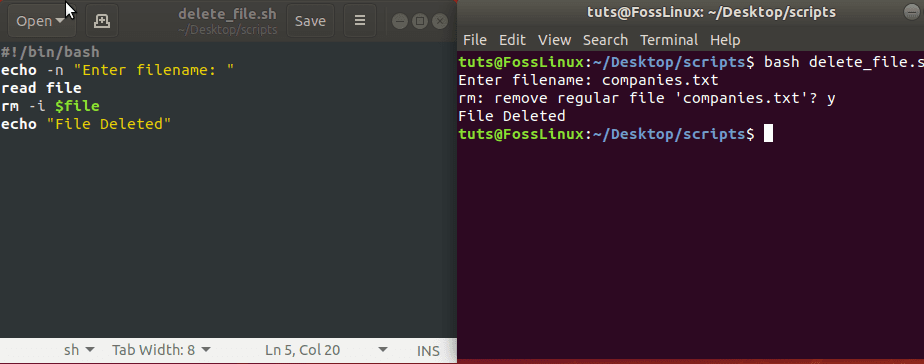
echo -n "Enter filename ->"

read name

rm -i $name

echo "File Deleted"

Output:



## Print Files With Line Count

In our example, we shall write a bash script that will print all files with there line count in the current directory.

#!/usr/bin/env bash

for F in \*

do

if [[ -f $F ]]

then

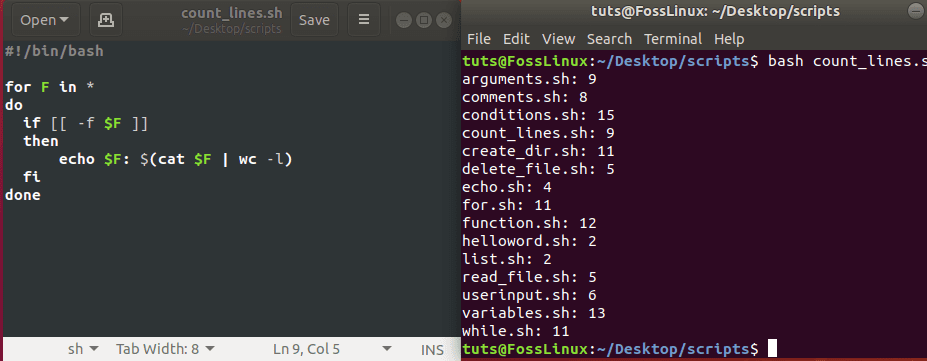
echo $F: $(cat $F | wc -l)

fi

done

You can see that we used a for loop to get the file and then used the [cat](https://www.fosslinux.com/27248/10-linux-cat-command-examples-for-beginners.htm) command to count lines.

Output:



## Print Number of Files and Folders

In our next example, the Linux bash script finds the number of files or folders present inside a given directory. It uses the Linux ‘find‘ command. Users will be asked to input the directory name where you want to search for files from the command-line.

#!/bin/bash

if [ -d "$@" ]; then

echo "Files found: $(find "$@" -type f | wc -l)"

echo "Folders found: $(find "$@" -type d | wc -l)"

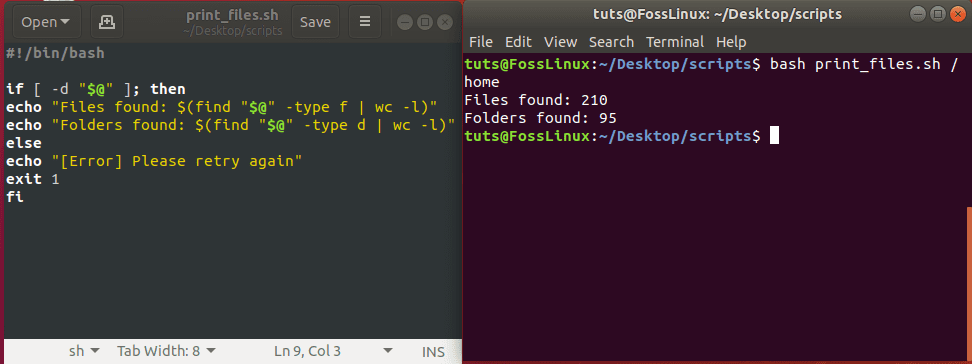
else

echo "[ERROR] Please try again."

exit 1

fi

Output:



## Error handling in Bash scripts

Let your Bash script help you find its errors with error handling.

Scripting is one of the key tools for a sysadmin to manage a set of day-to-day activities such as running backups, adding users/groups, installing/updating packages, etc. While writing a script, error handling is one of the crucial things to manage.

This article shows some basic/intermediate techniques of dealing with error handling in Bash scripting. I discuss how to obtain the error codes, get verbose output while executing the script, deal with the debug function, and standard error redirection. Using these techniques, sysadmins can make their daily work easy.

### Exit status

In Bash scripting, $? prints the exit status. If it returns zero, it means there is no error. If it is non-zero, then you can conclude the earlier task has some issue.

A basic example is as follows:

**$** cat myscript.sh

           #!/bin/bash

           mkdir learning

           echo $?

If you run the above script once, it will print 0 because the directory does not exist, therefore the script will create it. Naturally, you will get a non-zero value if you run the script a second time, as seen below:

**$** sh myscript.sh

mkdir: cannot create directory 'learning': File exists

1

### Best practices

It is always recommended to enable the debug mode by adding the -e option to your shell script as below:

**$** cat test3.sh

!/bin/bash

set -x

echo "hello World"

mkdiir testing

 ./test3.sh

+ echo 'hello World'

hello World

+ mkdiir testing

./test3.sh: line 4: mkdiir: command not found

You can write a debug function as below, which helps to call it anytime, using the example below:

**$** cat debug.sh

**#**!/bin/bash

\_DEBUG="on"

function DEBUG()

{

 [ "$\_DEBUG" == "on" ] && $@

}

DEBUG echo 'Testing Debudding'

DEBUG set -x

a=2

b=3

c=$(( $a + $b ))

DEBUG set +x

echo "$a + $b = $c"

Which prints:

**$** ./debug.sh

Testing Debudding

+ a=2

+ b=3

+ c=5

+ DEBUG set +x

+ '[' on == on ']'

+ set +x

2 + 3 = 5

### Standard error redirection

You can redirect all the system errors to a custom file using standard errors, which can be denoted by the number **2**. Execute it in normal Bash commands, as demonstrated below:

**$** mkdir users 2> errors.txt

**$** cat errors.txt

mkdir: cannot create directory ‘users’: File exists

## Error Handling

There is no try … catch blocks in bash for exception and error handling to say. So, how do you even start to handle errors in a way that none will escape and wreak havoc in the background hidden by silent a foreground that only appears okay.

Knowing how to use exit codes, options such as errexit, and traps allow you to create robust scripts and better handle errors in bash.

### **Exit codes**

Handling errors based on exit codes is the standstill method for detecting failure in a command. This is especially true in the case of external commands. Curl in bash is a good example of how to handle errors based on known error codes. Unlike user-defined functions, you can expect external command errors codes to be well documented.

${?} holds the exit code of the last command executed before any given line. An exit code of 0 means the command executed without any issues. Otherwise, something went wrong.

With error code you have a way to test if a command was successful or not.

What if you just want your bash script to die in the case that something goes wrong to minimize the damage caused by a script with errors?

That is where exit on error shows its precautious face.

### **Exit on error**

Exit on error unarguably the most useful feature enabling error detection and handling that bash programmers don’t start out with.

Create a script named minefield.sh:

#!/bin/bash  
## minefield

##################################################  
minefield() {  
a00075e82f2d59f3bd2b4de3d43c6206e50b93bd2b29f86ee0dfcb0012b6  
25ea27311a7503bd1698e2a111b5b61556f0495e84e09e098af281a0fc66  
39e62a385ca44b4bff58fb759b90c3f076fc2b0313e1d467d91984ea7965  
3517b2d1e5f9367c74c33c8a65749d3081a6f67da3567f408350289a60dc  
46ceb1b167a3cb71b90e96c1af5921bd241893d2bac18b56e93e70d43836  
66382963ecf70b1a664442262330a9a4d30b81076b1a2240019ee9c601b1  
4c88bc34634f9824daaaac2ca30f6c19bfb99cac9dcecd3cc1a30cc1e4c6  
bf75042e3ee312b4be98415841b3646ec3134cd549a25920c0628be11771  
c773f4e37b25a85b93124c6aec58b3e900b354bc8eff976b4da5835e2e44  
f7bc8317c368fb61350f1533f88173cb5d6f5f3f78323e2ee82d15e31480  
8b9f871c11a42282a7b68d7217708cf7a7554c4845a0f790bcf2b4b91f69  
c112ec610a54191aa3f6911dca73ea25ff987303cd5d57444ed6c9c2f815  
c310549929221426de23021ca1282d6cfc88169d0f0724d5b3a3069ca6c5  
54f362952413a46cfb9d3768caacc56781a02c47d2012b47fa25f54a9641  
1053161b6d05e1bcc3b08138d104d0cbef87cce54b19bb1f6f45ab99f283  
737364653dd1f7db292fd773b600bc6f6da8b28a10f0461befa59dac1436  
0fe9041a7ceb82b7fd307a6b53c09ad2234e889386d6b05b34fdf28d58ae  
4237033b6fe610e1b7e9f5b64503c3b82442604b888b40a6401e2a87390e  
43ced3824a2613ed0a8f684155baae5a4877b59be188845d692f8cb88626  
e9887620c9ab503ab3d550e0e8464482cb244d590483a88b9068695ecc16  
09f16eec806c3eef20775ef8071e56c9a190ab46eb32cbc8322c037265f2  
9fad2c9ee63acd67b2ede0dc2a33d5079e805e91b78f34c23d11bb7858cd  
b05043267e1e9b029ee06a29f18cdbbc79d9ff51758df378422e2d48f714  
d583894c014cb662ac69e7bcea19bab555eeacccd390c4900485e936d532  
1d7ff7e61a7c9208d401fcd64d4f6db14bf27c08316990145c6309bb457e  
fed43e657352a318331436d70168439b187106de19db544a63203ea32fd9  
1396bdde706d44dbc57d2ed2f8bafe4dfc2095d0aa9666b40d24b5ce1810  
339c4403a396950d2dd2c6096aac47faa35cb261e87709ddbf14b8a2cd4b  
968a582cc42a5aec7547067eb4cbd1656fd537d0d1f779d0739e12dfdefb  
0d8210092ffcf3f15b86a2b391e27ac9f9999c2a924b3116ffdfa939d48d  
202f8e3b90eec4c0c9eed6a5b0a2a481884ba12ebc1f5d77ecf7067a3d2d  
035efaee367ae2ada5c88565dad5296c6830ab81c1a27a9f298f07d243da  
878c245d100239424a92732aed3288b016fc3b6de668a95a87e61d74393e  
08097e063f318d20accf4952c65cca51399087f20f1737a006a3b8588e1a  
f022f3a3b4c3f2bf35d2888fe62fc026954c193a78d419da6f92a0e059b4  
aee037ed522d9c1c27b9610781cb502bdaa1a7c17816b8b546f5199f6260  
6799cbfbf0a3ae5f3b2b21c65abdf1e11a8055d9c12858fe88308fd3b5b1  
d61a115fb88542b02c888cb2325a6adc7473cd9308ca5d8368f5eccaf946  
5c534aaf1a13c5e2b721230096b641ded14379f52038728908ba23b4f7a1  
ce60c9c739115e72baf51ca840b956b4edfefa8e6709cd379f0b0d5be068  
fc4bdd73841954d95a38cbf4dea42814b39dba5cbb6d74c2319c8ddab86b  
7007acee1e3fa94a56c4ad6a233dfee5dac2e5b68963b917b9677ebf7a12  
ed4201c85ec976c313aad671a9b2a1bbd9e637e00ad980d0ab9ad343f5ae  
e36a95c20fc2f307bcfc1bd081ceb1ca28340b263924e7d31337d75a78ee  
b79dc0eedd74a34e4f05e613b275686f96ae32982e351c040637a614b376  
565acca108f821c99e33707a023379ef922e35b326081313207da19d8a73  
621ffcf4a7360a0647c2e6c9dd33496b1f048be5a5b4e7e9edf9fad06faa  
008545958e2544af6f645fb0d188baa0f61aeebf77e7fd78545fbe3d9313  
a65ed288ca87b0d41da534fa3e388138f0e78b0f494db0d0b7496745f83c  
a9314c0e4d988089db2e389ac0d23cb2e123269a826a19052fa955184b98  
85e026c5592197cd37377dd8c4052f1c86add447125db7d4c60010f7a42e  
9b8b550a88d2e5f61fe924925e982299d01c7670ebcda5c94b39860513c2  
44061da41147978c5d023722c839626fa522b546e998a148f22b61b851f2  
154ded671a1dab8350d8d48afccf9a8103eacf18067e203cb7270877fe9f  
8e87618811f9311c0b7898fb4a716216434ebc6ee296afcdcf5d60b069a1  
cfd5e3a86594fa56b8523c7102669742d7b6fea07550f37bdab67542c637  
a703575a24117adbc86a0daaccc7cb7cd313e6c7931d606fc3300b069a62  
44d588bab5b9d8c4f32c6fbe3a89a4e87a17db7df2802024cf0b66104460  
2055f71f4820bba44a4c86334c5dff744730df82ad9f7463522a618fa1c7  
561fff72366db9b918ac4c3cca86505163341a8e453774835e01495baf55  
cf9abf6f54fcf2f8f8a2d7e2f1b716ebf8842772639adfb0b878fbd353bd  
5295839f9806053498970cd093b39a09db0b06cb87a9b30861946f71287b  
2cc3ace9cf26fb75cb594d07924cdfb806b6cacda40de6dc0998c960ba62  
8006a6099065659be1d89bb8c2100569bd3f6d808968fa13a0208873da4e  
f1b62a2914c9fb9514ca5bcc8eddfbea54b12869fd1deb3f9eab9fd4d654  
7546db28931bee8ff44ad0f359775e95a1aeac6e752a3b35b9410932ed09  
66d307a834a1301f6622d249a98e99eb03bac2a569bbb3440cc6d8e7cc07  
9416d2616e2ee126b41fc5d350c33a036baa704aef70b01ace7fee2c62ff  
a480ec6141c9a2afc0f9fb2849570a69dba07dcfa70f8c78d11f61a2043f  
c81d3c3a594ec9f2d05bc9fb2514dd176cb9b16a712643a5a2808e33d8fa  
cb2faa622a38fab3d72e5eafe012912e8bed3ed5930f61c40c65df6fe644  
80d71eb9b825686e801cb27ade3ac2bf89ea63005f12d7c0ad51cd36d0f7  
6101821d0196ded179d90ad3e6601fcd46f11723adf30ce4c14b6495f94d  
d827422bd413386c10e118ee26fda1b27eda25bb93fc3dec5eda84771218  
8a6da66d1f6104bb565b7000439f4660b46f32d987e3890f4ddf56098dd2  
}  
##################################################  
if [ ${#} -eq 0 ]  
then  
 true  
else  
 exit 1 # wrong args

fi  
##################################################  
minefield

As you would expect, this is what happens when you try to run the script.

~$ ./minefield.sh

Output

minefield.sh: line 6: a00075e82f2d59f3bd2b4de3d43c6206e50b93bd2b29f86ee0dfcb0012b6:  
command not found  
minefield.sh: line 7: 25ea27311a7503bd1698e2a111b5b61556f0495e84e09e098af281a0fc66:  
command not found  
minefield.sh: line 8: 39e62a385ca44b4bff58fb759b90c3f076fc2b0313e1d467d91984ea7965:  
command not found  
minefield.sh: line 9: 3517b2d1e5f9367c74c33c8a65749d3081a6f67da3567f408350289a60dc:  
command not found  
minefield.sh: line 10: 46ceb1b167a3cb71b90e96c1af5921bd241893d2bac18b56e93e70d43836:  
command not found  
minefield.sh: line 11: 66382963ecf70b1a664442262330a9a4d30b81076b1a2240019ee9c601b1:  
command not found  
:

Fun. But would it be nice to stop after hitting the first mine? There is a way to do that in bash, i.e. exit on error. Let’s try it again with errexit set on.

~$ ./minefield.sh