BASH SCRIPTING

Bash Scripting Tutorial with Examples

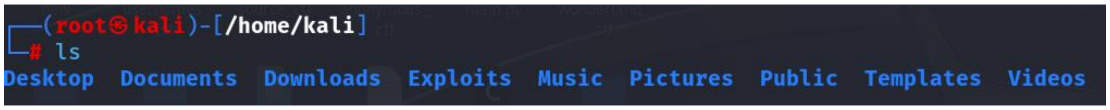
Emre Koşar

Table of Contents

What is Bash?

Bash stands for ***Bourne-Again Shell***. It is a command line interpreter and widely used in GNU/Linux OS. Bash is one of the most popular shell environment. Shell refers to a program that provides a command-line interface for interacting with an OS. Shell allows us to enter commands and then it returns us with the appropriate results of provided commands as output.

To give an idea, here is a simple example:



```
┌──(root㉿kali)-[/home/kali]
└─# ls
Desktop  Documents  Downloads  Exploits  Music  Pictures  Public  Templates  Videos
```

As we can see on the picture above, a command (*ls*) has been entered and as result of this process, we are able to see the output of that command. In case of entering an incorrect command, we'd get an error as ***command not found***. That's because Shell won't be recognizing the random commands.

Why Should We Use Bash Scripting?

Bash scripting can lead to some advantages such as automating routine tasks in Unix/Linux based systems. There are additional advantages of using Bash Scripting. Here are some of them:

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

Portability: Shell scripts can be run on various platforms and operating systems, including Unix/Linux/macOS and even Windows.

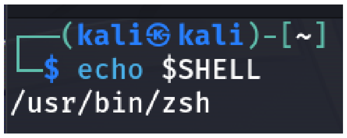
Debugging: Shell scripts are easy to debug, and most shells have built-in-debugging and error reporting tools that can help identify and fix issues quickly.

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

Introduction to Bash Scripting

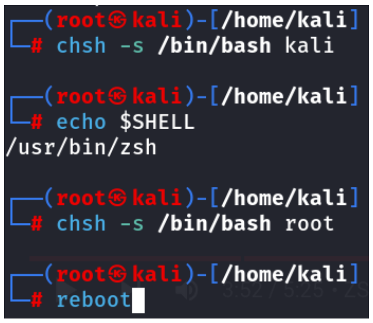
Switching to Bash

In order to learn and practice Bash Scripting, you are required to have Linux installed and make sure that you're running Bash as Shell. To see which Shell is running our system, we can simply enter '***echo $SHELL***' or '***echo $0***' command.



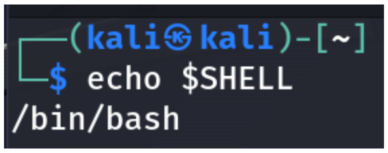
```
┌──(kali㉿kali)-[~]
└─$ echo $SHELL
/usr/bin/zsh
```

As you can see, I'm running '***Z shell***' which is one another Shell environment. Let's switch to the Bash.



```
┌──(root㉿kali)-[/home/kali]
└─# chsh -s /bin/bash kali

┌──(root㉿kali)-[/home/kali]
└─# echo $SHELL
/usr/bin/zsh

┌──(root㉿kali)-[/home/kali]
└─# chsh -s /bin/bash root

┌──(root㉿kali)-[/home/kali]
└─# reboot
```

To switch to Bash from Z Shell, we need to use "***chsh***" command. The '***chsh***' command stands for 'change shell'. Syntax would be '***chsh -s <shell path> username***'. If you noticed, after applying the command, Shell environment was still the same. That's because **reboot** is required in order to save the changes.

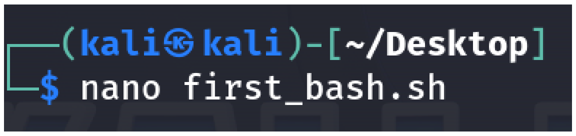


```
┌──(kali㉿kali)-[~]
└─$ echo $SHELL
/bin/bash
```

After rebooting, you can confirm that shell successfully switched to Bash.

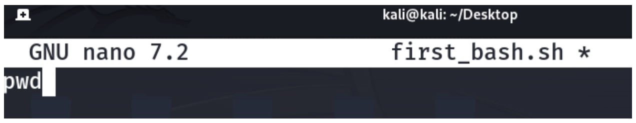
Creating Our First Bash File

When creating a Bash File, we will be following the same path as creating random file. We can either use '*touch*' or '*nano*' commands. The thing to pay attention to here is the file extension. We need to name our file as format of 'my_file***.sh***'.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ nano first_bash.sh
```

As you can see, I've created a Bash file named '*first_bash.sh*'. Let's write something simple inside of that file in order to understand the logic of Bash files.

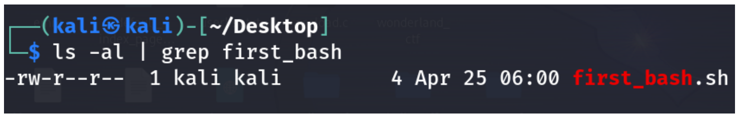


```
kali@kali: ~/Desktop
  GNU nano 7.2                    first_bash.sh *
pwd
```

As you can see, the '***pwd***' (print working directory) command is written inside of the Bash file. To save it inside nano text editor, follow the steps below:

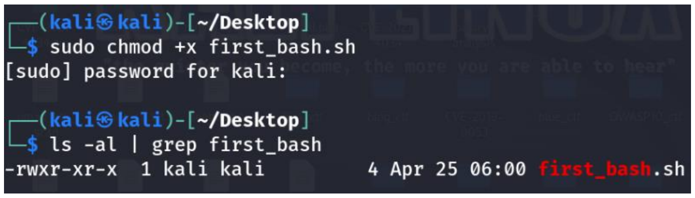
- Press CTRL+O,
- Press ENTER,
- Press CTRL+X.

The file has been saved. To execute the Bash File, we need to give executable permission to it.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ls -al | grep first_bash
-rw-r--r--  1 kali kali          4 Apr 25 06:00 first_bash.sh
```

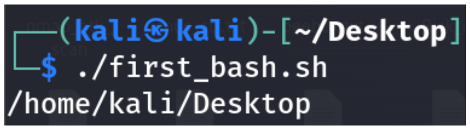
As you can see, the file itself doesn't have the executable permission. To give executable permission, simply use '***chmod +x your_file_name.sh***' command.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ sudo chmod +x first_bash.sh
[sudo] password for kali:

┌──(kali㉿kali)-[~/Desktop]
└─$ ls -al | grep first_bash
-rwxr-xr-x  1 kali kali          4 Apr 25 06:00 first_bash.sh
```

After giving executable permission, we can control if it's done or not by using '***ls -l***' command. If we look at the output of the second command on the left-hand side, executable permission is successfully given.

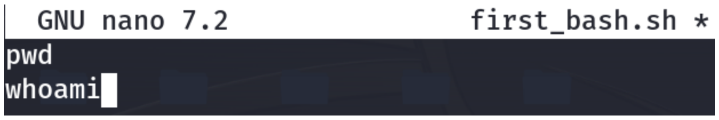
If you come so far, you can run the file you've created.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
/home/kali/Desktop
```

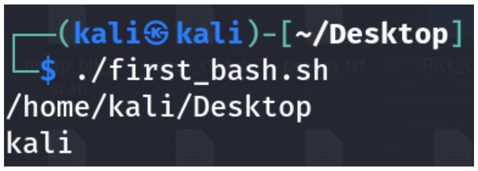
The program returned pwd command's result.

We can add many more commands into Bash file we desire. For instance:



```
  GNU nano 7.2                    first_bash.sh *
pwd
whoami
```

I've added '***whoami***' command and that Bash file will return me the results of both commands.



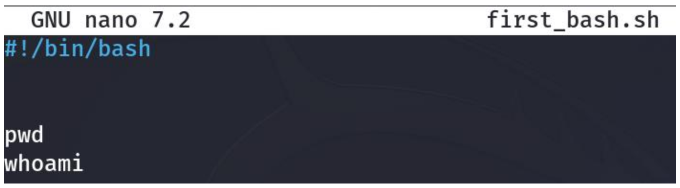
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
/home/kali/Desktop
kali
```

Since there were two different commands executed on the script, output of each command would be on different lines.

The two commands in the Bash file don't represent the actual structure of Bash script. These were meant to be understand the logic. In addition, Bash files can have if statements, loops and things like that. I'll be demonstrating how to control our Bash file according to our points.

Properly Writing Bash Script

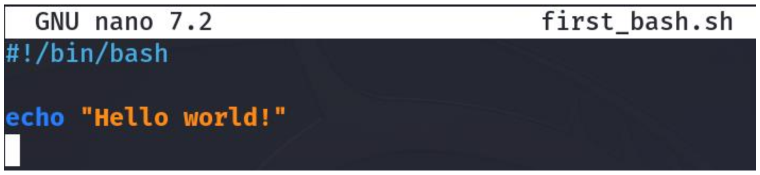
First of all, we need to implement a line called ***Shebang Line***. The Shebang is the combination of the '*#*' and '*!*'. It is used to specify the interpreter with which the given script will be run by default. The Shebang Line has an important role in Shell scripting, especially when dealing with different types of shell.



```
  GNU nano 7.2                                first_bash.sh
#!/bin/bash


pwd
whoami
```

Here is Shebang Line implemented on our simple example.

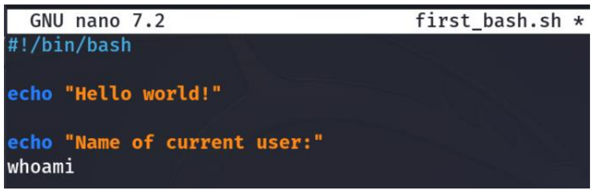
Now, let's start deep diving with another simple example.



```
  GNU nano 7.2                              first_bash.sh
#!/bin/bash

echo "Hello world!"

```

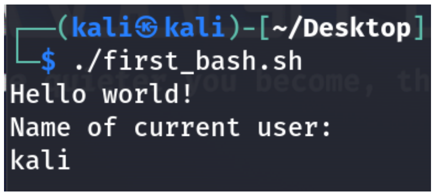
As you can guess, this script will return '***Hello world!***'

Let's add one more thing. I will be adding something to show how to combine echo (print) and another shell command.



```
  GNU nano 7.2                                      first_bash.sh *
#!/bin/bash

echo "Hello world!"

echo "Name of current user:"
whoami
```

Since two different '*echo*' commands executed, both sentences are expected to be printed on different lines. Hence, '*pwd*' is one another command itself, it's also printed on next line.



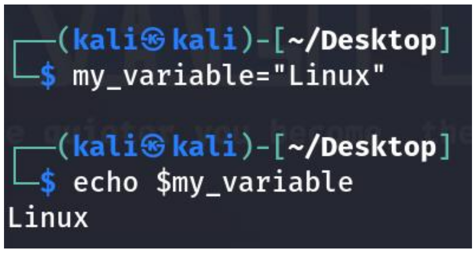
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Hello world!
Name of current user:
kali
```

As you can tell from the output, each line of scripts printed on different lines.

Variables in Bash Scripting

Variables are used in programming languages in general. The purpose of using variables is to hold values or data. These assigned values can change during the execution of a program, hence the term variable.

In Bash Scripting, declaring a variable is simple. I'll demonstrate an example.



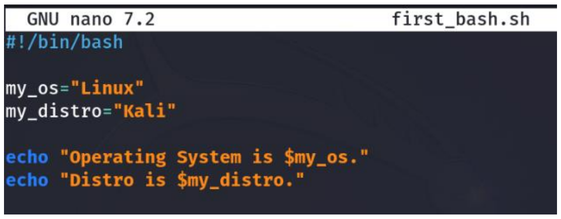
```
┌──(kali㉿kali)-[~/Desktop]
└─$ my_variable="Linux"

┌──(kali㉿kali)-[~/Desktop]
└─$ echo $my_variable
Linux
```

In the first command, variable has been declared. In the second command, variables have been printed onto screen.

- Whenever we reference a variable in Bash, we have to put a Dollar sign in front of the name of the variable. Otherwise, it's not going to work exactly as we might expect.
- Let's say a previously undeclared variable name has taken part on the 'echo' line. Bash will return empty (null) result.
- Also keep in mind that if you declared variables in terminal and once you exit the terminal and reopened it you can't 'echo' them because they are all gone since the session has closed.

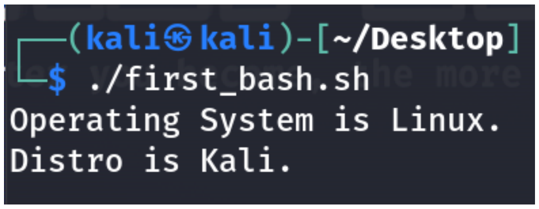
Using Variables in Bash Script Files

I will demonstrate an easy example of using variables in our Bash File.



```
  GNU nano 7.2                               first_bash.sh
#!/bin/bash

my_os="Linux"
my_distro="Kali"

echo "Operating System is $my_os."
echo "Distro is $my_distro."
```

If you pay attention to the example, there is a Dollar sign in front of the variable again. So, it doesn't matter if we're on the terminal or not; we have to use it if variables need to be mentioned.

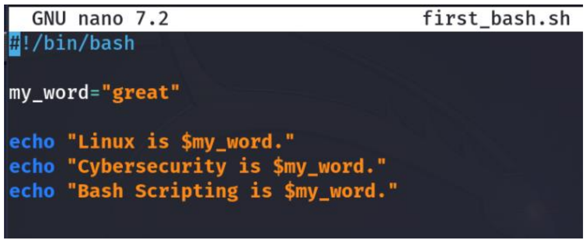


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Operating System is Linux.
Distro is Kali.
```

Values corresponding to variables were printed on the output where they placed on the script.

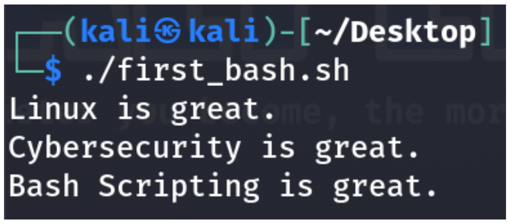
- One more thing to pay attention is that we need to place variables in between double quotes. Instead, if we placed in between single quotes, Bash would ignore the value corresponding the variable, it'd print the variable name itself.

Examples



```
GNU nano 7.2                                  first_bash.sh
#!/bin/bash

my_word="great"

echo "Linux is $my_word."
echo "Cybersecurity is $my_word."
echo "Bash Scripting is $my_word."
```

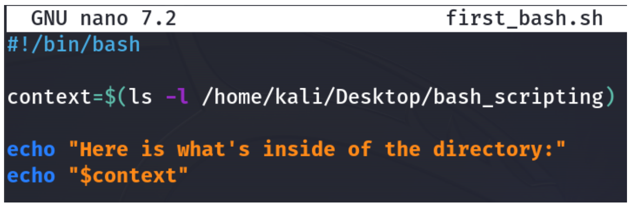
In this example, a variable called '*my_word*' has assigned to '***great***' value. The purpose is to not write 'great' in each line recursively. This type of usage also gives the opportunity to change the word in one shot.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Linux is great.
Cybersecurity is great.
Bash Scripting is great.
```

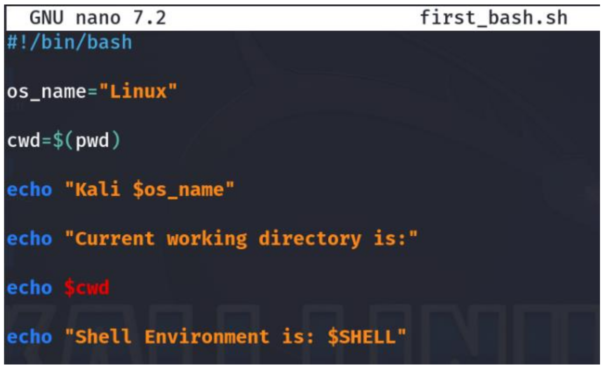
'***great***' value has been printed at where its variable placed on the script.

What if we want to store the output of a shell command inside of a variable?



```
GNU nano 7.2                                  first_bash.sh
#!/bin/bash

context=$(ls -l /home/kali/Desktop/bash_scripting)

echo "Here is what's inside of the directory:"
echo "$context"
```

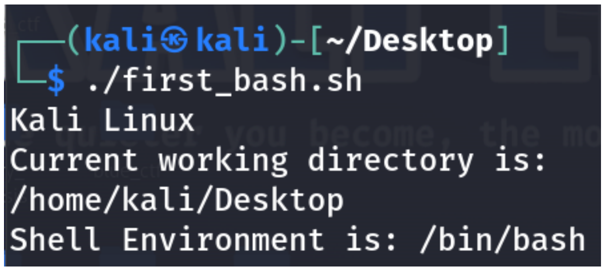
The corresponding value/method to 'context' variable is called *subshell*. (***$(command)***) Subshell allows us to execute a command in the background. Speaking of the script, I've declared a variable named 'context' and let the '*ls -l /home/kali/Desktop/bash_scripting*' command to work in the background. After that, I printed the output by using '*echo*'.



```
GNU nano 7.2                                  first_bash.sh
#!/bin/bash

os_name="Linux"

cwd=$(pwd)

echo "Kali $os_name"

echo "Current working directory is:"

echo $cwd

echo "Shell Environment is: $SHELL"
```

In this example, '***os_name***' variable has assigned to '***Linux***' and '***cwd***' variable has assigned to output of 'pwd' command. On the last line, as you can see after the Dollar sign, there is *$SHELL* variable however we didn't declare it. You would think that we might face an error.

- The ***$SHELL*** variable is one of the default variables in our Shell environment. There are many more default variables such as ***$USER*** , ***$OSTYPE*** etc. These kinds of variables are always declared. Please make sure that the variables you declare do not have the same name as these default variables.



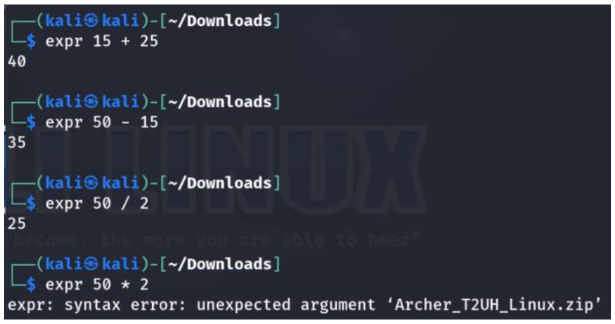
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Kali Linux
Current working directory is:
/home/kali/Desktop
Shell Environment is: /bin/bash
```

That's the output of the example. As you can see, *the* ***$SHELL*** variable returned */bin/bash* value.

<u>HINT:</u> If you want to see default variables in your Shell environment, you can run the '***env***' command on your terminal.

Mathematical Operations in Bash Scripting

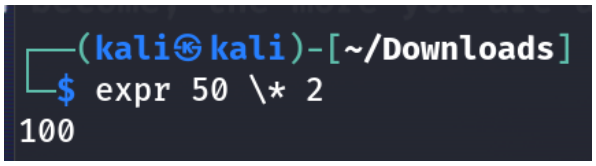
Mathematical operations in Bash Scripting could be useful in some cases. Such as converting CPU temperature to Fahrenheit. Syntax of mathematical operations in Bash Scripting is a bit different than other programming languages. In Bash Scripting, there is an evaluate the expressions command which is '***expr***'.



```
┌──(kali㉿kali)-[~/Downloads]
└─$ expr 15 + 25
40

┌──(kali㉿kali)-[~/Downloads]
└─$ expr 50 - 15
35

┌──(kali㉿kali)-[~/Downloads]
└─$ expr 50 / 2
25

┌──(kali㉿kali)-[~/Downloads]
└─$ expr 50 * 2
expr: syntax error: unexpected argument 'Archer_T2UH_Linux.zip'
```

The '***expr***' command would run summation, subtraction and division. But when there is multiplication, it won't work. Because asterisk (*) in Bash is an important thing and it means *everything*. Asterisk symbol runs the command against everything in local directory.

To avoid this type of misuse, use the ***escape operator***. (\)

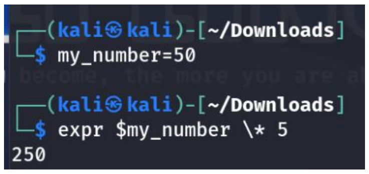


```
┌──(kali㉿kali)-[~/Downloads]
└─$ expr 50 \* 2
100
```

Thanks to the escape operator, Bash understands the asterisk symbol as multiplication.

Using Variables in Mathematical Operations

In this section, we will talk about how to use our pre-defined variables (by us) in Mathematical Operations. It's actually simple and similar to what we have covered in previous sections.

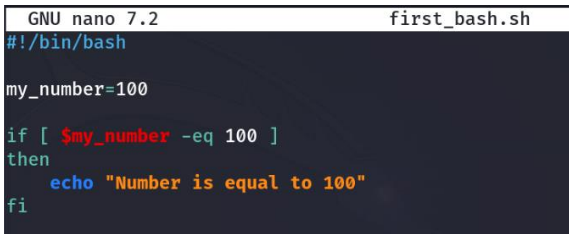


```
┌──(kali㉿kali)-[~/Downloads]
└─$ my_number=50

┌──(kali㉿kali)-[~/Downloads]
└─$ expr $my_number \* 5
250
```

Firstly, the variable has been defined and then simply placed its name after Dollar sign.

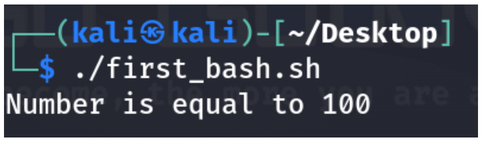
If Statement in Bash Scripting

If Statement is one of fundamental construct for decision making structure in programming. In Bash Scripting, syntax of If Statement is a little bit different. The If Statement must be closed with '*fi*' (reversed of if) after the instructions under the statement. In order to give an overview of the if statements, there is a simple example below.



```
  GNU nano 7.2                  first_bash.sh
#!/bin/bash

my_number=100

if [ $my_number -eq 100 ]
then
    echo "Number is equal to 100"
fi
```

As how it's defined above, if statements end with '***fi***' keyword. The '***-eq***' represents ***equal***. Nevertheless, you can free to use equal sign instead of '***-eq***' (single equal sign not double). One more thing is indentations. You have to pay attention to indentations like I did in the example.



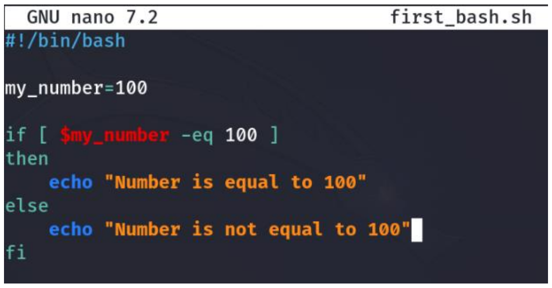
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Number is equal to 100
```

Since the defined number equals 100, hence condition is provided, we can see the output. If the condition was false, the program would return empty result.

HINT: If you're coding on terminal (not on IDE) and have to indent a Tab, don't press Tab button directly. Instead, press the Space button 4 times.

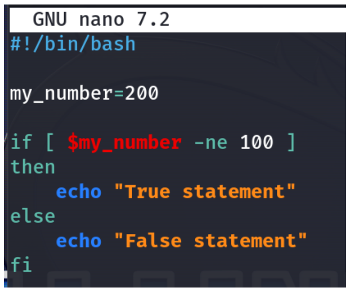
Creating If-Else Conditional Blocks in Bash Scripting

Now, let's go further. If '*else*' statement also used in conditions, the '*fi*' keyword have to be placed under else condition. So, there is no need to place '*fi*' under if condition. As an instance, I will insert an Else condition to previous script.



```
  GNU nano 7.2                  first_bash.sh
#!/bin/bash

my_number=100

if [ $my_number -eq 100 ]
then
    echo "Number is equal to 100"
else
    echo "Number is not equal to 100"
fi
```

As you can see, the '***fi***' keyword has been moved from under if block to else block. That's because the condition doesn't end after if block.

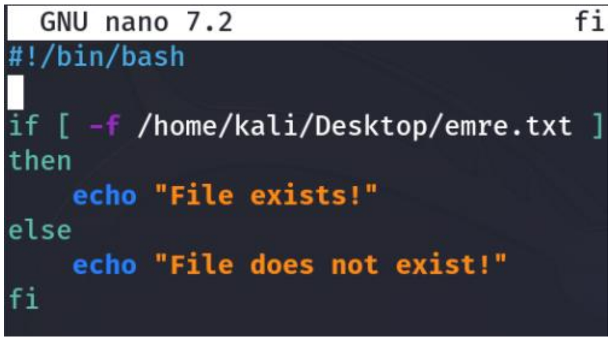


```
GNU nano 7.2
#!/bin/bash

my_number=200

if [ $my_number -ne 100 ]
then
    echo "True statement"
else
    echo "False statement"
fi
```

This example returns '***True statement***' if '*my_number*' is **not** equal to 100. Based on this short explanation, we can simply tell that '***-ne***' does the same thing as ' **!=** '. By the way, instead of using '***-eq***' or '***-ne***', '=' and '*!*=' can be used.

Checking for Existence of a File

This topic might be used on real life scenarios. As the topic name suggests, we will learn how to check if a file exists or not by using '*if-else*' statement.

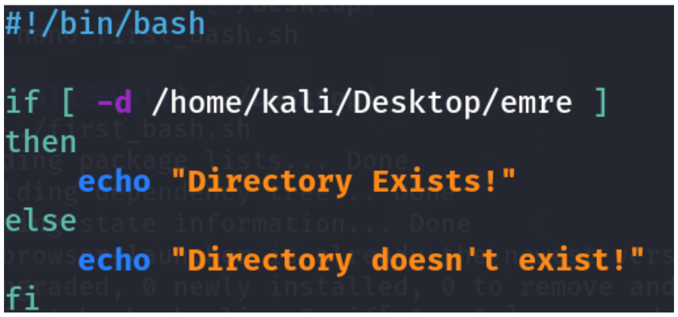


```
GNU nano 7.2                                    fi
#!/bin/bash

if [ -f /home/kali/Desktop/emre.txt ]
then
    echo "File exists!"
else
    echo "File does not exist!"
fi
```

The '***-f***' parameter represents checking for a file. You can memorize it from '**file**'s initial. The '***if***' block checks if there is a file named 'emre.txt' under /home/kali/Desktop directory. If it exists, 'File exists!' will be printed. If not, 'File does not exist!' will be printed.

Checking for Existence of a Directory

Maybe you can guess the parameter used to check for existence of a directory by looking at the previous section.

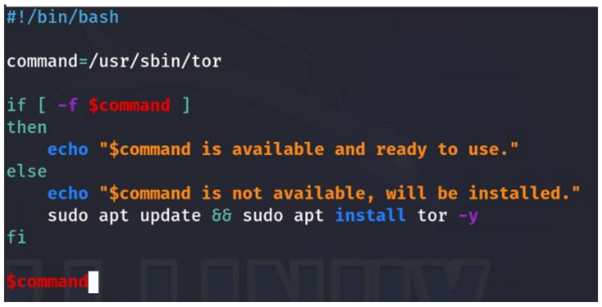


```
#!/bin/bash

if [ -d /home/kali/Desktop/emre ]
then
    echo "Directory Exists!"
else
    echo "Directory doesn't exist!"
fi
```

The '***-d***' parameter represents checking for a directory. You can also memorize it from '**directory**'s initial. The 'if' block checks if there is a directory named 'emre'.

Creating a Bash Script File to Install Package

In this section, we'll write a script which will install the package if it's not installed on the system yet. This one might also be useful as automating the process on real life scenarios.

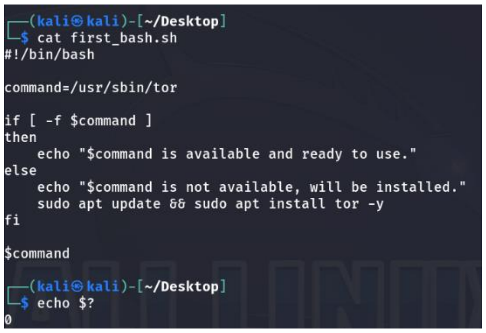


```
#!/bin/bash

command=/usr/sbin/tor

if [ -f $command ]
then
    echo "$command is available and ready to use."
else
    echo "$command is not available, will be installed."
    sudo apt update && sudo apt install tor -y
fi

$command
```

Let's demonstrate line by line. First, the location of utility has been declared (to find out you can run *which utility_name*). If it's installed on the system, only information pops out, nothing will trigger. However, if it's not installed, information pops out and utility (tor) will be installed. Once the if-else statement is over, command will be triggered.

Exit Codes in Bash Scripting

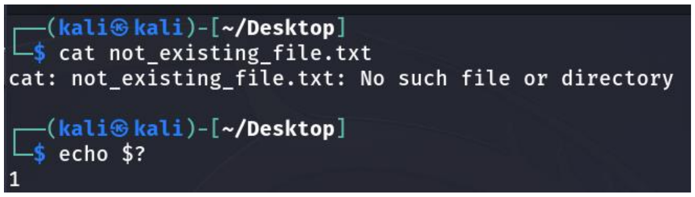
Each Shell command returns an exit code when it terminates, either successfully or unsuccessfully. Basically, a certain condition is met, and script should have stopped and terminated. This is the case exit codes come to play. An exit code of '**zero**' indicates that the command completed ***successfully***, and **non-zero** means that an ***error*** was encountered. The special variable ***$?*** returns the exit status of the last executed command.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ cat first_bash.sh
#!/bin/bash

command=/usr/sbin/tor

if [ -f $command ]
then
    echo "$command is available and ready to use."
else
    echo "$command is not available, will be installed."
    sudo apt update && sudo apt install tor -y
fi

$command

┌──(kali㉿kali)-[~/Desktop]
└─$ echo $?
0
```

As you can see, we tried to read an existing file on my system and didn't have any error messages. To verify it, we can print the exit status as in the last command of the example script by using '***echo $0***'. As mentioned, zero represents no error.

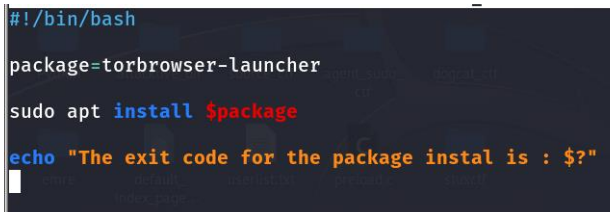
In case of running an invalid command or looking for non-existing file/directory, obviously we'll get error. So, the exit status code won't be zero. Here is how it looks for particular case:



```
┌──(kali㉿kali)-[~/Desktop]
└─$ cat not_existing_file.txt
cat: not_existing_file.txt: No such file or directory

┌──(kali㉿kali)-[~/Desktop]
└─$ echo $?
1
```

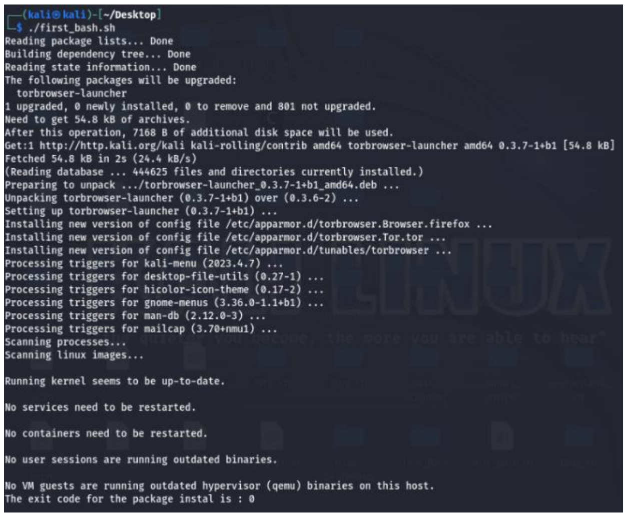
- Why do we need exit status codes?

Assume that you have a task that you are supposed to do tomorrow. But you want it to be finished by the night, so you have written a script to autonomous the task. The script might fail in some particular parts. When you wake up, you'll see where the problem is and take action.



```
#!/bin/bash

package=torbrowser-launcher

sudo apt install $package

echo "The exit code for the package instal is : $?"
```

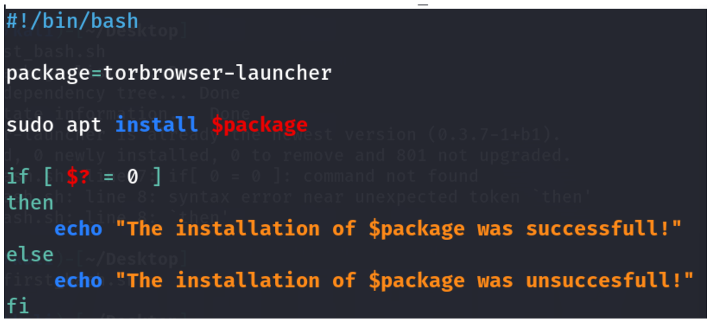
This script is meant to install '*package*' variable which is '***torbrowser-launcher***' in this case.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages will be upgraded:
  torbrowser-launcher
1 upgraded, 0 newly installed, 0 to remove and 801 not upgraded.
Need to get 54.8 kB of archives.
After this operation, 7168 B of additional disk space will be used.
Get:1 http://http.kali.org/kali kali-rolling/contrib amd64 torbrowser-launcher amd64 0.3.7-1+b1 [54.8 kB]
Fetched 54.8 kB in 2s (24.4 kB/s)
(Reading database ... 444625 files and directories currently installed.)
Preparing to unpack .../torbrowser-launcher_0.3.7-1+b1_amd64.deb ...
Unpacking torbrowser-launcher (0.3.7-1+b1) over (0.3.6-2) ...
Setting up torbrowser-launcher (0.3.7-1+b1) ...
Installing new version of config file /etc/apparmor.d/torbrowser.Browser.firefox ...
Installing new version of config file /etc/apparmor.d/torbrowser.Tor.tor ...
Installing new version of config file /etc/apparmor.d/tunables/torbrowser ...
Processing triggers for kali-menu (2023.4.7) ...
Processing triggers for desktop-file-utils (0.27-1) ...
Processing triggers for hicolor-icon-theme (0.17-2) ...
Processing triggers for gnome-menus (3.36.0-1.1+b1) ...
Processing triggers for man-db (2.12.0-3) ...
Processing triggers for mailcap (3.70+nmu1) ...
Scanning processes...
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
The exit code for the package instal is : 0
```

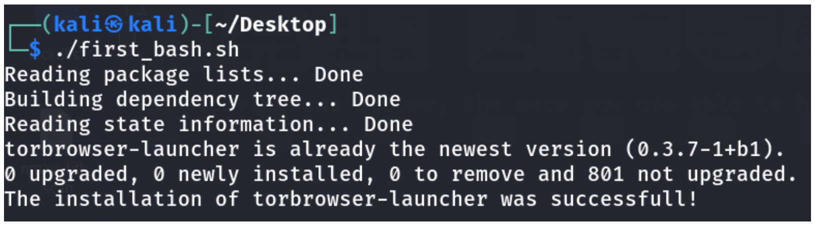
After executing the script, *torbrowser-launcher* was installed on the system, which means there were no errors. At the bottom of the output (last line), you can see that the exit code is zero.

Let's combine exit status code with if-else statement in a script.



```
#!/bin/bash

package=torbrowser-launcher

sudo apt install $package

if [ $? = 0 ]
then
       echo "The installation of $package was successfull!"
else
       echo "The installation of $package was unsuccesfull!"
fi
```

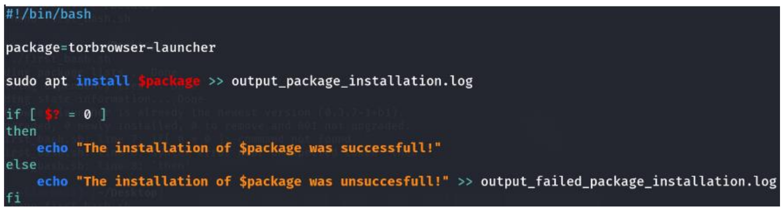
The purpose of this script is if exit status code equals zero, which means successful, echo the message above '*if*' block. If exit status code equals non-zero, echo the message above '*else*' block.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
torbrowser-launcher is already the newest version (0.3.7-1+b1).
0 upgraded, 0 newly installed, 0 to remove and 801 not upgraded.
The installation of torbrowser-launcher was successfull!
```

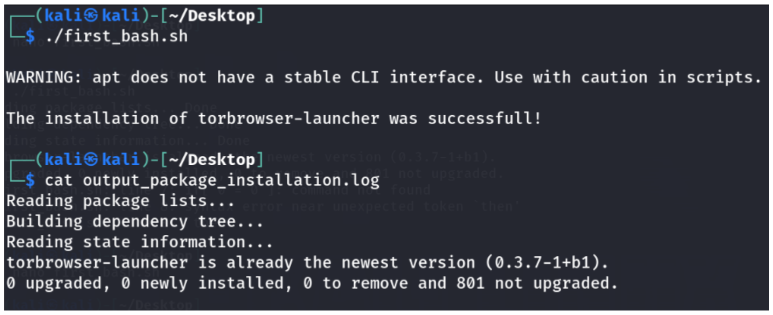
As you can see from the bottom of the output, it was successfully installed.

Based on the recent script, let's get closer to real-life script by modifying it.



```
#!/bin/bash

package=torbrowser-launcher

sudo apt install $package >> output_package_installation.log

if [ $? = 0 ]
then
    echo "The installation of $package was successfull!"
else
    echo "The installation of $package was unsuccesfull!" >> output_failed_package_installation.log
fi
```

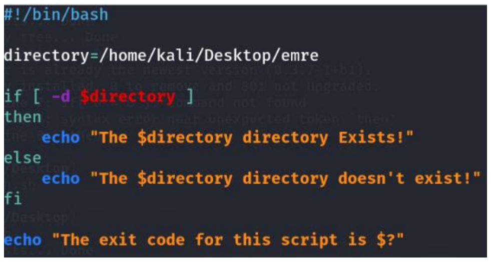
The two greater than symbols (>>) are redirecting the command into something we obtain. Rather than printing all of the information on the screen, what I did by using '>>' was to redirect that output into file.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh

WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

The installation of torbrowser-launcher was successfull!

┌──(kali㉿kali)-[~/Desktop]
└─$ cat output_package_installation.log
Reading package lists...
Building dependency tree...
Reading state information...
torbrowser-launcher is already the newest version (0.3.7-1+b1).
0 upgraded, 0 newly installed, 0 to remove and 801 not upgraded.
```

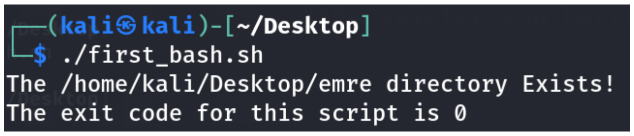
As of now, output looks clearer. All of the process is written in the log file as you can see.

Examples



```
#!/bin/bash

directory=/home/kali/Desktop/emre

if [ -d $directory ]
then
    echo "The $directory directory Exists!"
else
    echo "The $directory directory doesn't exist!"
fi

echo "The exit code for this script is $?"
```

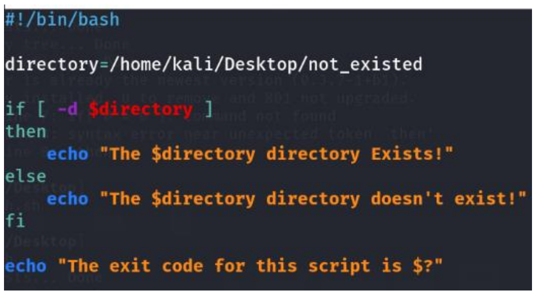
The purpose of this script is to see if the provided directory exists or not. To do that if-else statement is used. By the way, the directory specified in this example exists the system.



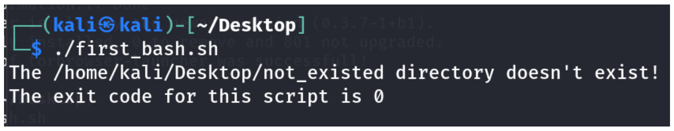
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
The /home/kali/Desktop/emre directory Exists!
The exit code for this script is 0
```

As you can see, we can verify that the directory exists on the system. So, the exit status code returned as zero.

What if there was a non-existed directory provided? Let's find out.



```
#!/bin/bash

directory=/home/kali/Desktop/not_existed

if [ -d $directory ]
then
    echo "The $directory directory Exists!"
else
    echo "The $directory directory doesn't exist!"
fi

echo "The exit code for this script is $?"
```



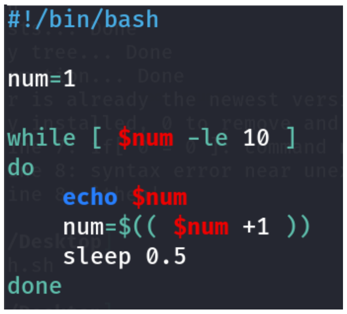
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
The /home/kali/Desktop/not_existed directory doesn't exist!
The exit code for this script is 0
```

Even though directory doesn't exist on the system, the exit status code returned as zero again. Let's break it down.

Reason why we got exit code of zero is because as you can remember, '***$?***' gives the most recent command's status. Since the most recent command executed on the system is '***echo***' under if-else statement, exit status code will be zero.

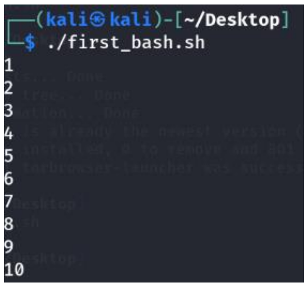
While Loop in Bash Scripting

While Loop is one of the fundamental components of programming. A While Loop is a statement that iterates over a block of code until the particular condition is met. Syntax of While Loop in Bash Scripting is not that different than other programming languages. Let's break it down in an example.



```
#!/bin/bash
num=1
while [ $num -le 10 ]
do
    echo $num
    num=$(( $num +1 ))
    sleep 0.5
done
```

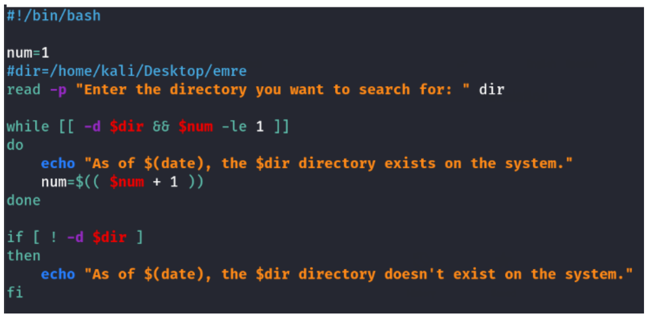
First things first, a variable called '***num***' has assigned as 1. In 'while loop', the variable will be printed, and it will increase by 1 until it reaches 10 (-le parameter means less or equal <=). Inside the loop, our variable '*num*' Basically, the script will count to 10 when it's executed. Additionally, '***sleep 0.5***' was added inside the loop for modern view.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
1
2
3
4
5
6
7
8
9
10
```

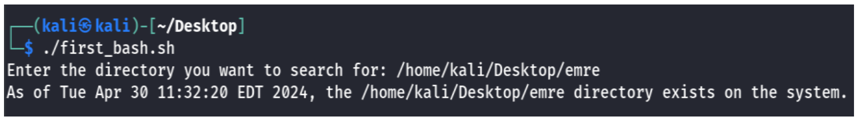
As expected, the script counted all the way up to 10. When it hits 10, the script terminates itself.

Let's do one more example. In the subsequent example, there will be fresh commands and methods.



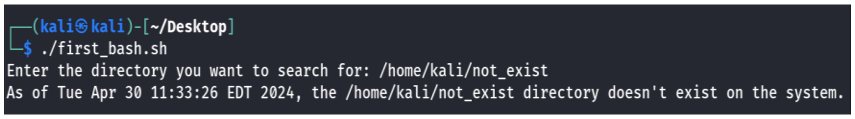
```
#!/bin/bash

num=1
#dir=/home/kali/Desktop/emre
read -p "Enter the directory you want to search for: " dir

while [[ -d $dir && $num -le 1 ]]
do
    echo "As of $(date), the $dir directory exists on the system."
    num=$(( $num + 1 ))
done

if [ ! -d $dir ]
then
    echo "As of $(date), the $dir directory doesn't exist on the system."
fi
```

Let's try to understand the script line by line. First, to prevent infinite loop, we used '***num***' variable inside the while loop as it's defined as 1. Instead of giving the directory inside the script, we would go with getting it from user. In order to do that, we used '***read***' command which is used for getting input. On top of that, '***-p***' parameter is used for printing the provided message and after the message '***dir***' is input variable. If we are looking to while loop, there are two conditions provided. What they are responsible for is to check if the directory exists or not and in order to prevent infinite loop, to check '***num***' variable to see if it's less or equal to 1. The purpose of 'if' statement is to inform that directory does not exist on the system. It only works when the user gives an invalid directory.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Enter the directory you want to search for: /home/kali/Desktop/emre
As of Tue Apr 30 11:32:20 EDT 2024, the /home/kali/Desktop/emre directory exists on the system.
```

This is what the output looks like when a valid directory is checked.

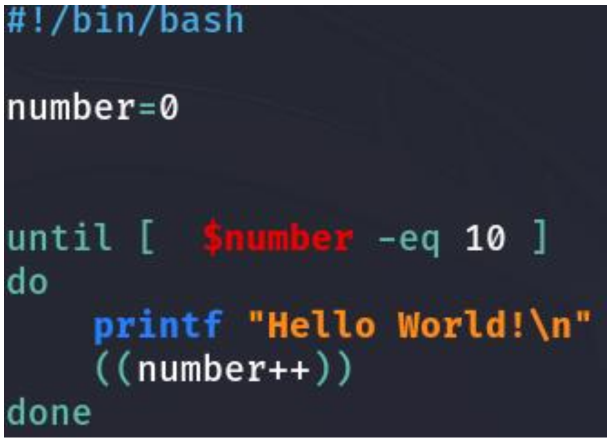


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Enter the directory you want to search for: /home/kali/not_exist
As of Tue Apr 30 11:33:26 EDT 2024, the /home/kali/not_exist directory doesn't exist on the system.
```

This one is what the output looks like when an invalid directory is checked.

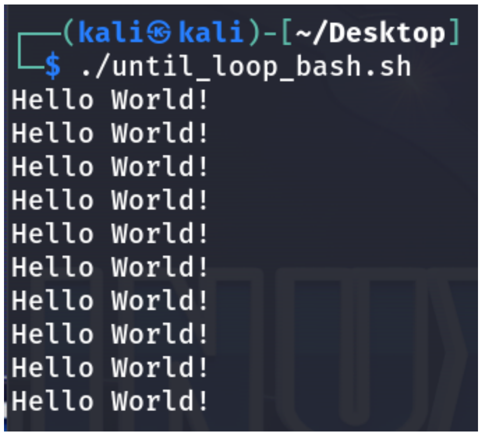
Until Loop in Bash Scripting

In Bash scripting, there is not just '*while*' loop. In Bash, there are types of loops that we can use. They are '*while*', '*until*' and '*for*' loops. As the topic suggests, we will look into the '*until*' loop. The syntax of '*until*' loop is similar to while loop. Let's break it down in example.



```
#!/bin/bash

number=0


until [  $number -eq 10 ]
do
    printf "Hello World!\n"
    ((number++))
done
```

As you can see, the syntax of '*until*' loop is similar to '*while*' loop. This script will return '***Hello World!***' until the number which is defined as zero reaches 10 increasing by 1 in each execution. Once it reaches 10, the script would stop working immediately. Additionally, there is a keyword that we didn't use previously is used in this script, which is '***printf***'. The '*printf*' command has almost the same function as the '*echo*' command. The difference between them is the '*echo*' command sends new line at the end of the output, unlike '*printf*' which we have to use '**\n**' option to create new line after the message.

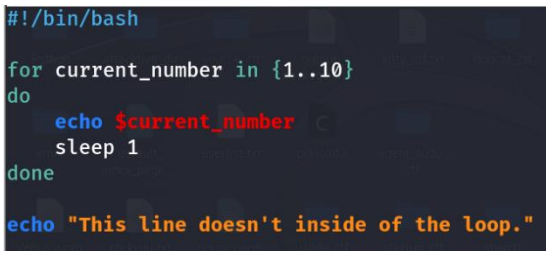


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./until_loop_bash.sh
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

On the output, '***Hello World!***' printed ten times on the screen. That's because the loop stopped working after printed '*Hello World!*' for tenth times and the number became greater than ten.

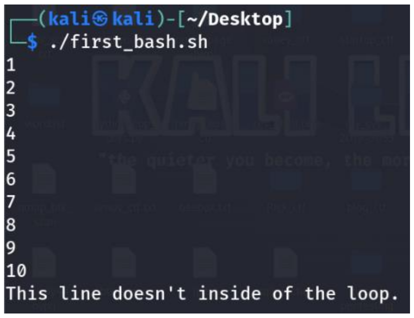
For Loop in Bash Scripting

For Loop is also one of the core components of programming. By using For Loops in our scripts, we can perform repetitive tasks and tricky problems in a simple programmatic way. Speaking of the syntax of For Loops in Bash Scripting, there are two types of usage of For Loops. C-style For Loop and POSIX-style For Loop. The syntax of POSIX-style similar to Syntax of For Loops in Python. In general, POSIX-style is preferred for its readability and flexibility, but both of the styles serve the same purpose. Let's see it in the examples.



```bash
#!/bin/bash

for current_number in {1..10}
do
    echo $current_number
    sleep 1
done

echo "This line doesn't inside of the loop."
```

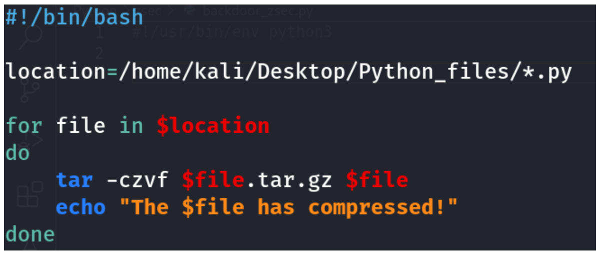
The logic of the script is the same as the first example of 'While Loop' above. Inside of the for loop, '***current_number***' variable will equal whatever the current item iterated in the loop. The '***{1..10}***' represents 1 to 10 array, so for loop will print the numbers by order. Additionally, '***sleep 1***' was added inside the loop for modern view.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
1
2
3
4
5
6
7
8
9
10
This line doesn't inside of the loop.
```

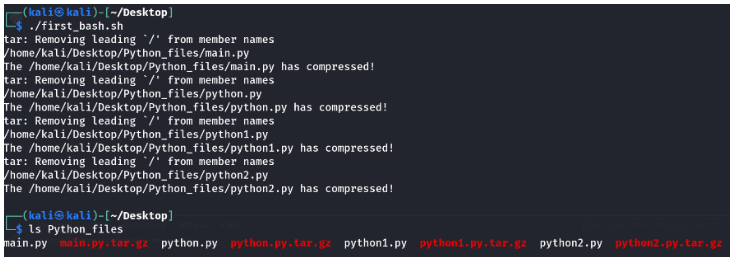
The script printed from 1 to 10 and the echo line as it's supposed to be.

Let's write a useful script for the real-life scenarios.



```bash
#!/bin/bash

location=/home/kali/Desktop/Python_files/*.py

for file in $location
do
    tar -czvf $file.tar.gz $file
    echo "The $file has compressed!"
done
```

In this script, any files that end with '***.py***' inside the specific location will be compressed. After compressing each one of the files, the operation will be printed.



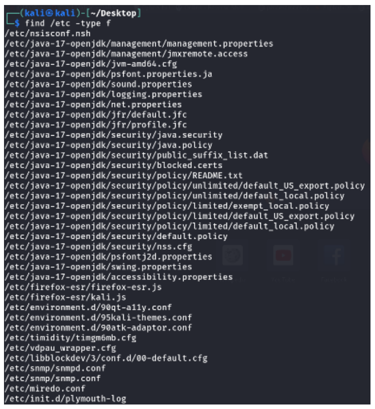
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
tar: Removing leading `/' from member names
/home/kali/Desktop/Python_files/main.py
The /home/kali/Desktop/Python_files/main.py has compressed!
tar: Removing leading `/' from member names
/home/kali/Desktop/Python_files/python.py
The /home/kali/Desktop/Python_files/python.py has compressed!
tar: Removing leading `/' from member names
/home/kali/Desktop/Python_files/python1.py
The /home/kali/Desktop/Python_files/python1.py has compressed!
tar: Removing leading `/' from member names
/home/kali/Desktop/Python_files/python2.py
The /home/kali/Desktop/Python_files/python2.py has compressed!

┌──(kali㉿kali)-[~/Desktop]
└─$ ls Python_files
main.py  main.py.tar.gz  python.py  python.py.tar.gz  python1.py  python1.py.tar.gz  python2.py  python2.py.tar.gz
```

As the image proves, all of the files under the '***Python_files***' directory had been compressed.

Data Streams

In general, the definition of Data streaming is the continuous transfer of data at a high rate of speed. Data streams are collecting data from various data sources. There are three types of Data Streams in computer programming. In previous sections, there is one type of data stream that is highly used, which is ‘***STDOUT***’ (Standard Output). For instance, ‘pwd’ or any other commands that print something on the screen are example of Standard Output. Let’s talk about the other type of data stream. For instance, let’s assume that you wanted to list a non-existed directory and you typically get error. That’s called ‘***STDERR***’ (Standard Output). The last type of data stream is called ‘***STDIN***’ (Standard Input). It’s used when the program reads its input data.

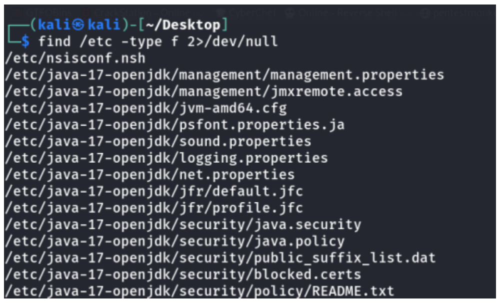
Let’s clarify the standard error and standard output by using the ‘*find*’ command.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ find /etc -type f
/etc/nsisconf.nsh
/etc/java-17-openjdk/management/management.properties
/etc/java-17-openjdk/management/jmxremote.access
/etc/java-17-openjdk/jvm-amd64.cfg
/etc/java-17-openjdk/psfont.properties.ja
/etc/java-17-openjdk/sound.properties
/etc/java-17-openjdk/logging.properties
/etc/java-17-openjdk/net.properties
/etc/java-17-openjdk/jfr/default.jfc
/etc/java-17-openjdk/jfr/profile.jfc
/etc/java-17-openjdk/security/java.security
/etc/java-17-openjdk/security/java.policy
/etc/java-17-openjdk/security/public_suffix_list.dat
/etc/java-17-openjdk/security/blocked.certs
/etc/java-17-openjdk/security/policy/README.txt
/etc/java-17-openjdk/security/policy/unlimited/default_US_export.policy
/etc/java-17-openjdk/security/policy/unlimited/default_local.policy
/etc/java-17-openjdk/security/policy/limited/exempt_local.policy
/etc/java-17-openjdk/security/policy/limited/default_US_export.policy
/etc/java-17-openjdk/security/policy/limited/default_local.policy
/etc/java-17-openjdk/security/default.policy
/etc/java-17-openjdk/security/nss.cfg
/etc/java-17-openjdk/psfontj2d.properties
/etc/java-17-openjdk/swing.properties
/etc/java-17-openjdk/accessibility.properties
/etc/firefox-esr/firefox-esr.js
/etc/firefox-esr/kali.js
/etc/environment.d/90qt-a11y.conf
/etc/environment.d/95kali-themes.conf
/etc/environment.d/90atk-adaptor.conf
/etc/timidity/timgm6mb.cfg
/etc/vdpau_wrapper.cfg
/etc/libblockdev/3/conf.d/00-default.cfg
/etc/snmp/snmpd.conf
/etc/snmp/snmp.conf
/etc/miredo.conf
/etc/init.d/plymouth-log
```

The command is ‘***find /etc -type f***’ which means find any files under ‘**/etc**’ directory. The point to keep in mind is that the command will print both STDOUT and STDERR. In this case, standard errors would be ‘Permission Denied’ alert since the command executed as user and user can’t see what’s inside of some files. So, if the command was executing as root, there would be no error for this case.

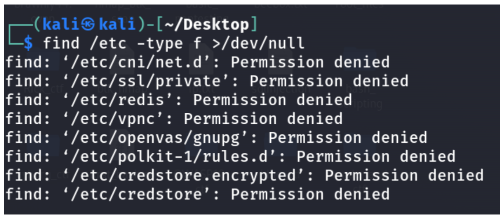
If we don’t want to see any errors, we can distinguish standard error and standard output. Also keep in mind that this could be useful while writing a Bash Script automation or something like that.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ find /etc -type f 2>/dev/null
/etc/nsisconf.nsh
/etc/java-17-openjdk/management/management.properties
/etc/java-17-openjdk/management/jmxremote.access
/etc/java-17-openjdk/jvm-amd64.cfg
/etc/java-17-openjdk/psfont.properties.ja
/etc/java-17-openjdk/sound.properties
/etc/java-17-openjdk/logging.properties
/etc/java-17-openjdk/net.properties
/etc/java-17-openjdk/jfr/default.jfc
/etc/java-17-openjdk/jfr/profile.jfc
/etc/java-17-openjdk/security/java.security
/etc/java-17-openjdk/security/java.policy
/etc/java-17-openjdk/security/public_suffix_list.dat
/etc/java-17-openjdk/security/blocked.certs
/etc/java-17-openjdk/security/policy/README.txt
```

Adding ‘**2>/dev/null**’ ignores the standard errors and prints only the standard output. So, there would be no single error if ‘2>/dev/null’ parameter is used. What this ‘***2>/dev/null***’ parameter does is to redirect the standard errors into /dev/null which is a special place in Linux file system that wipes out everything in particular place. However, if you want to see the standard errors, you can redirect into some another place.

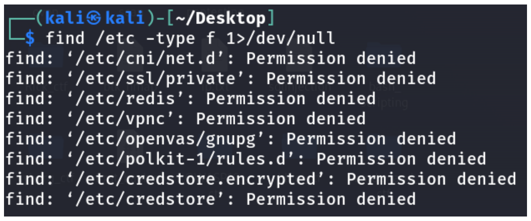
What if we use nothing or 1 instead of 2 while redirecting? Let’s find out.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ find /etc -type f >/dev/null
find: '/etc/cni/net.d': Permission denied
find: '/etc/ssl/private': Permission denied
find: '/etc/redis': Permission denied
find: '/etc/vpnc': Permission denied
find: '/etc/openvas/gnupg': Permission denied
find: '/etc/polkit-1/rules.d': Permission denied
find: '/etc/credstore.encrypted': Permission denied
find: '/etc/credstore': Permission denied
```

This command sends standard output to /dev/null. Since the errors didn’t send anywhere in particular, they will be printed on the screen as you can see.

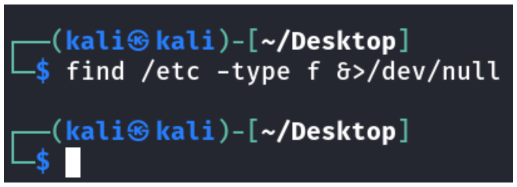
Let’s try ‘1>/dev/null’.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ find /etc -type f 1>/dev/null
find: ‘/etc/cni/net.d’: Permission denied
find: ‘/etc/ssl/private’: Permission denied
find: ‘/etc/redis’: Permission denied
find: ‘/etc/vpnc’: Permission denied
find: ‘/etc/openvas/gnupg’: Permission denied
find: ‘/etc/polkit-1/rules.d’: Permission denied
find: ‘/etc/credstore.encrypted’: Permission denied
find: ‘/etc/credstore’: Permission denied
```

As you can see, this type of redirection returns standard errors as well. So, **number 1** contains the output of the command which is standard output and **number 2** contains the error messages of executed command which is standard error. Since these numbers are redirected into a file, we see the opposite of redirected number as output.

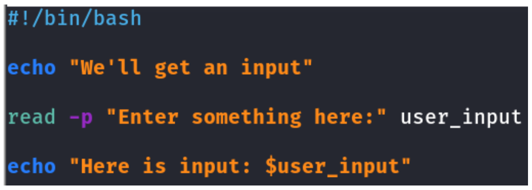
What if we want to redirect both standard output and standard errors into a particular place? Let’s find out.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ find /etc -type f &>/dev/null

┌──(kali㉿kali)-[~/Desktop]
└─$ 
```

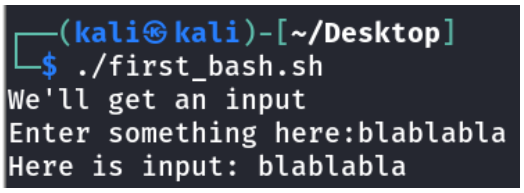
To send both standard output and standard errors, the **Ampersand (&)** symbol is used. Since both standard output and standard errors are redirected to a particular place (in this case it’s /dev/null), there is nothing on the output.

Enough talking about standard output and standard errors. Let’s have a look at the standard input (STDIN). As its name suggests, standard input is actually getting information from the user. Let’s take a look at an example.



```
#!/bin/bash

echo "We'll get an input"

read -p "Enter something here:" user_input

echo "Here is input: $user_input"
```

As the ‘***read***’ command and also ‘***-p***’ parameter previously used in examples to get user input, it was representing standard input. As a quick reminder, the ‘***read***’ command is used for getting input and the ‘***-p***’ parameter is used for printing a message on the screen before getting input.



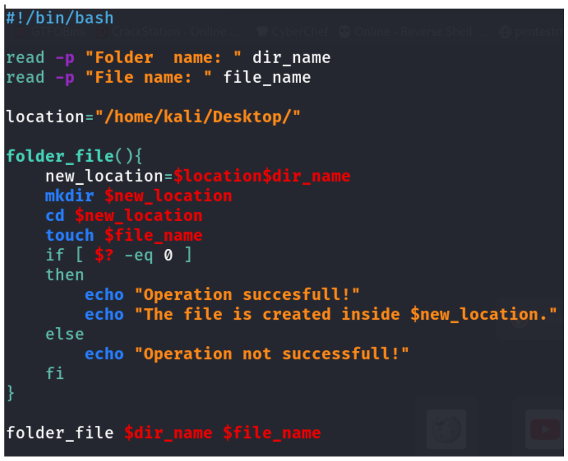
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
We'll get an input
Enter something here:blablabla
Here is input: blablabla
```

As mentioned above, the message printed before taking input.

Functions in Bash Scripting

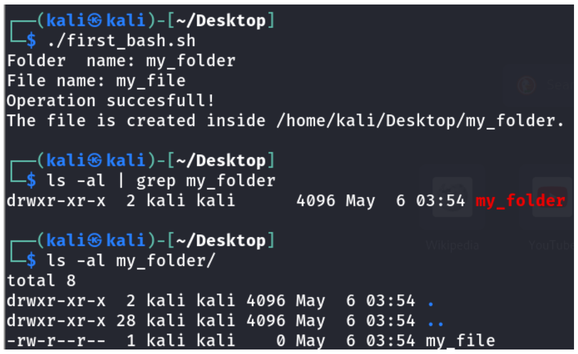
Functions are one of the key components in computer programming in general. Function allows us to reuse the set of commands. Think of a function as a small script within a script. It is a small chunk of code which can be called multiple times when it's needed.

Let's take a look at the example.



```
#!/bin/bash

read -p "Folder  name: " dir_name
read -p "File name: " file_name

location="/home/kali/Desktop/"

folder_file(){
    new_location=$location$dir_name
    mkdir $new_location
    cd $new_location
    touch $file_name
    if [ $? -eq 0 ]
    then
        echo "Operation succesfull!"
        echo "The file is created inside $new_location."
    else
        echo "Operation not successfull!"
    fi
}

folder_file $dir_name $file_name
```

Let's break it down line by line. First, user provide folder and file name separately by order. In this case, the name of the function is '***folder_file***'. Inside the function, the folder and the file are being created on Desktop. After then, if the exit status code equals to zero, 'operation successful' message will be printed on the screen. At the very bottom of the page, it can be seen that the function was called. The important point is '***dir_name***' and '***file_name***' variables were added right after the function name as passing arguments.

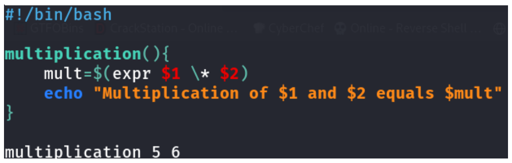


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./first_bash.sh
Folder  name: my_folder
File name: my_file
Operation succesfull!
The file is created inside /home/kali/Desktop/my_folder.

┌──(kali㉿kali)-[~/Desktop]
└─$ ls -al | grep my_folder
drwxr-xr-x  2 kali kali      4096 May   6 03:54 my_folder

┌──(kali㉿kali)-[~/Desktop]
└─$ ls -al my_folder/
total 8
drwxr-xr-x  2 kali kali 4096 May  6 03:54 .
drwxr-xr-x 28 kali kali 4096 May  6 03:54 ..
-rw-r--r--  1 kali kali    0 May  6 03:54 my_file
```

After running the script, it first asks for folder name and then the file name. Since the operation is successful, in other words, exit status code equals zero, it printed out the '**Operation successful**' message and the information message.

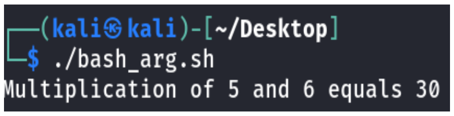
Passing Arguments to Function

In some cases, we might need to pass arguments to the function. What it does is actually sending data to the function to process when we pass arguments. The arguments are supposed to be specified right after calling the function. Let's create a small script in order to fully understand how it works.



```
#!/bin/bash

multiplication(){
    mult=$(expr $1 \* $2)
    echo "Multiplication of $1 and $2 equals $mult"
}

multiplication 5 6
```

In this script, the point is to pass the numbers as arguments right after the function and see if it's working or not. As a quick reminder, inside of the function, while defining a multiplication, we have to use asterisk (*), and it means '**ALL**' in Bash scripting. So, in order to specify that it's a mathematical operation, back slash (\) must be used before asterisk (*).

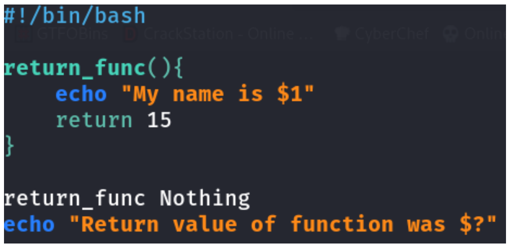


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_arg.sh
Multiplication of 5 and 6 equals 30
```

As the output proves, the function worked the way we wanted.

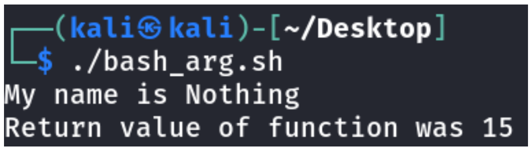
Return Values

A return value is produced and returned to the calling method by a function after it completes its execution. In computer programming, the concept of return value for functions is used to send data back to the original calling location. However, Bash doesn't allow us to do this. It allows us to set a return status code. In Bash scripting, the return value is assigned to '***$?***' variable. Let's see it in example.



```
#!/bin/bash

return_func(){
    echo "My name is $1"
    return 15
}

return_func Nothing
echo "Return value of function was $?"
```

Let's break the script down. Inside of the function, the return status was assigned to 15. So, in the output, we will see the return status value. In addition, the return status might be variable.

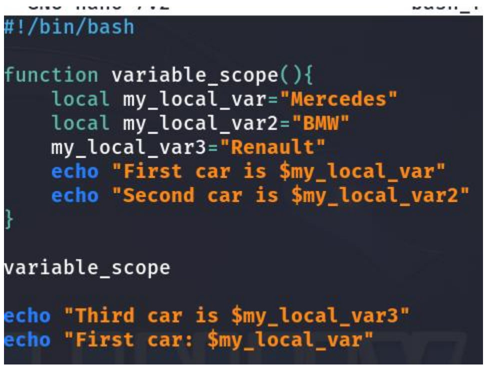


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_arg.sh
My name is Nothing
Return value of function was 15
```

As you can see, the '***$?***' variable was returned as 15 as obtained inside the function.

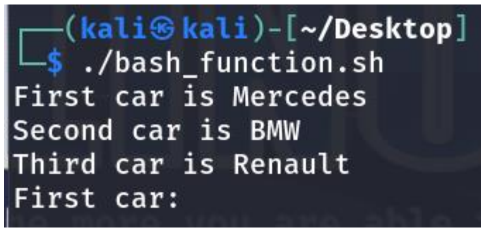
Variable Scopes

A variable is '***global***' by default. Global variable allows us to use the variable wherever we'd like inside the script file. When we create a variable inside of a function, it's a '*local*' variable, which can only be used inside of that function. To create a local variable, we have to use '*local*' keyword in front of the variable name itself. Let's see it in the example.



```
#!/bin/bash

function variable_scope(){
    local my_local_var="Mercedes"
    local my_local_var2="BMW"
    my_local_var3="Renault"
    echo "First car is $my_local_var"
    echo "Second car is $my_local_var2"
}

variable_scope

echo "Third car is $my_local_var3"
echo "First car: $my_local_var"
```

Inside of the function, a few variables were defined. However, pay attention that two of the variables have '***local***' keyword before defining them. That particular keyword makes those variables usable only inside of the function. So, outside of the function, calling these two variables would return nothing. But since the '***my_local_var3***' didn't define as local variable, it is callable outside of the function.

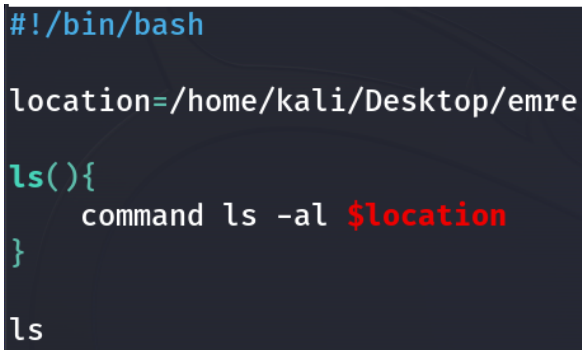


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_function.sh
First car is Mercedes
Second car is BMW
Third car is Renault
First car:
```

As you can see, in the last line of the output '***my_local_var***' didn't return anything since it's defined as local variable. But the line which '***my_local_var3***' variable was called worked correctly even though it's also defined inside the function. The reason is '***local***' keyword.

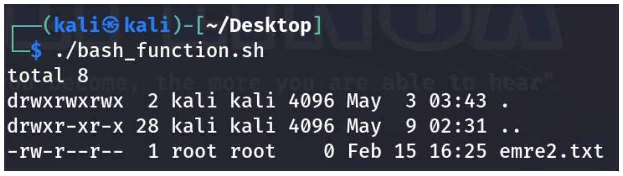
Overriding Commands

In Bash scripting, it is possible to name a function as the same name as a Bash command. In case of naming the function with a Bash command, that particular command will execute the orders inside the defined function, not the default orders. Let's say we want to run '***ls -la***' but we forget the parameters every time. If we could just create a function name '***ls***' inside of a Bash file and let it execute '***ls -la***' every time the function called, we would have achieved this. Let's check how it works.



```
#!/bin/bash

location=/home/kali/Desktop/emre

ls(){
    command ls -al $location
}

ls
```

Inside of the function, there is a keyword called '***command***'. When we have a function with the same name as a command, we have to put the '***command***' keyword in front of the command itself. Otherwise, it won't work.

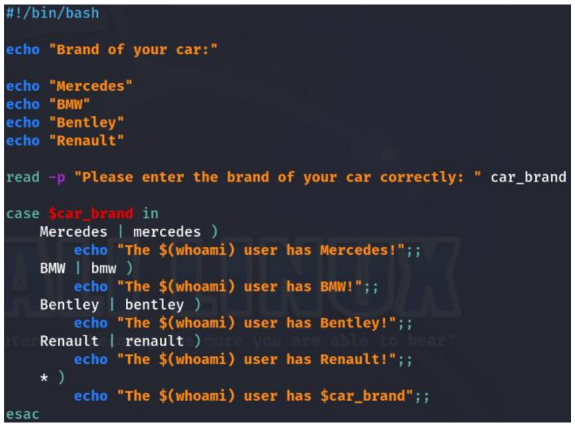


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_function.sh
total 8
drwxrwxrwx  2 kali kali 4096 May  3 03:43 .
drwxr-xr-x 28 kali kali 4096 May  9 02:31 ..
-rw-r--r--  1 root root    0 Feb 15 16:25 emre2.txt
```

As you can see, the function lists all files of */home/kali/Desktop/emre* directory.

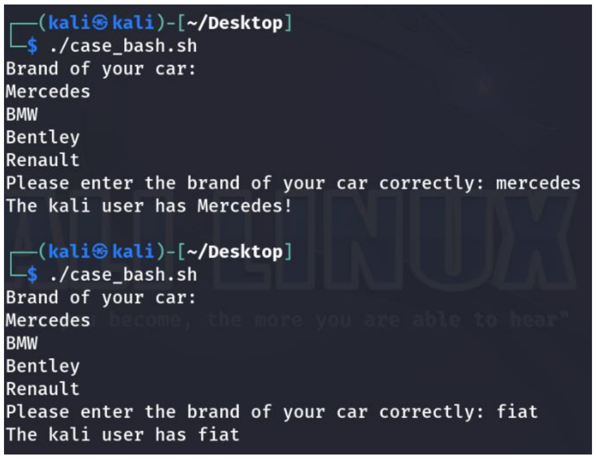
Case Statements in Bash Scripting

A case statement in Bash script is used when a decision has to be made against multiple choices. It's useful when there are multiple choices needed. Using the case statement instead of nested if statements would look more readable and reduce the complexity of the script. Additionally, the case statement has a concept similar to JavaScript or C Switch statement. However, the main difference between C switch statement and Bash case statement is that Bash Case statement doesn't continue to search for a pattern match once it has found one and executed statements associated with that pattern unlike C Switch statement. Let's take a look at how the case statement works.



```
#!/bin/bash

echo "Brand of your car:"

echo "Mercedes"
echo "BMW"
echo "Bentley"
echo "Renault"

read -p "Please enter the brand of your car correctly: " car_brand

case $car_brand in
    Mercedes | mercedes )
        echo "The $(whoami) user has Mercedes!";;
    BMW | bmw )
        echo "The $(whoami) user has BMW!";;
    Bentley | bentley )
        echo "The $(whoami) user has Bentley!";;
    Renault | renault )
        echo "The $(whoami) user has Renault!";;
    * )
        echo "The $(whoami) user has $car_brand";;
esac
```

In this script, the case statement will return a sentence depending on input from the user. So, the output sentence will vary based on the input. Let's break the case statement down. As variable, '***car_brand***' variable given for this case which is the user's input defined above. The Point of the case statement is to find out which of the case matches with the user input. The pipe (|) symbol separates the two options for a case. The last case inside of the statement will be executed only if the user input is different than the other defined cases. To finish the case statement, we must use '***esac***' which is reversed of case.



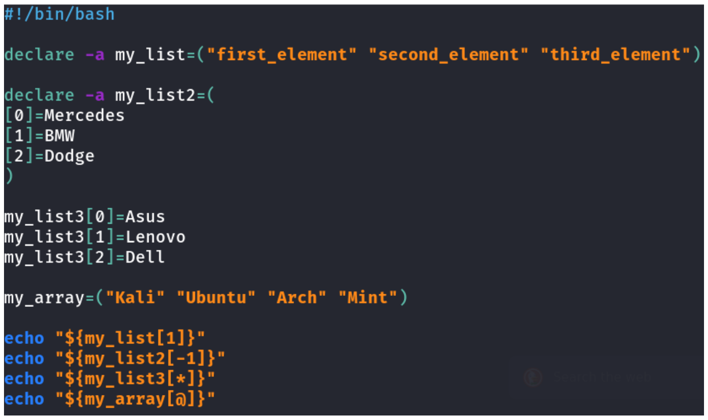
```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./case_bash.sh
Brand of your car:
Mercedes
BMW
Bentley
Renault
Please enter the brand of your car correctly: mercedes
The kali user has Mercedes!

┌──(kali㉿kali)-[~/Desktop]
└─$ ./case_bash.sh
Brand of your car:
Mercedes
BMW
Bentley
Renault
Please enter the brand of your car correctly: fiat
The kali user has fiat
```

In the first execution of the script, I gave '***mercedes***' as input. So, that triggers the specific case created for '**Mercedes**' or '**mercedes**' input. In the second execution of the script, I gave a brand which is not declared under the case statement. That one triggers the case created for all of the inputs except the given ones as cases.

Arrays in Bash Scripting

Arrays have an important role in programming in general. Arrays allow us to store and retrieve elements in a list form which can be used for certain tasks. In Bash, we can also have arrays that help us in creating scripts in the command line for storing data in a list format. While creating an array in Bash scripting, we have several options. One of the option is by using '*declare -a*' command before naming the array itself. One of another option is by manually assigning elements to array. Let's take a look at both ways in the example.

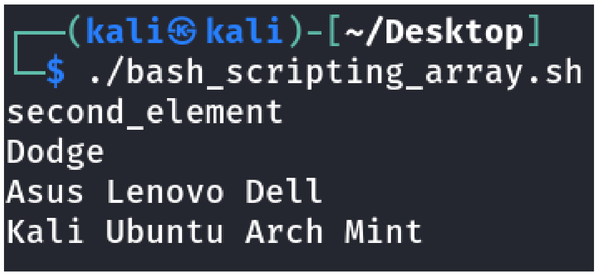
Creating and Printing Array



```
#!/bin/bash

declare -a my_list=("first_element" "second_element" "third_element")

declare -a my_list2=(
[0]=Mercedes
[1]=BMW
[2]=Dodge
)

my_list3[0]=Asus
my_list3[1]=Lenovo
my_list3[2]=Dell

my_array=("Kali" "Ubuntu" "Arch" "Mint")

echo "${my_list[1]}"
echo "${my_list2[-1]}"
echo "${my_list3[*]}"
echo "${my_array[@]}"
```

In this script, there are couple of arrays defined via different ways. Let's break it down one by one. In the first two arrays defined by using '***declare -a***' command. '***-a***' parameter is needed when we would like to define array. In the second array, ***my_list2*** , the order of elements is declared manually. The way third array, ***my_list3*** , defined is done by hand, row by row. The last array, ***my_array*** , is defined similar to defining a variable, just write the elements inside of the parentheses.

Let's talk about how to print the arrays. As you remember, when printing the variables, we have used '$' before the name of the variable. For arrays, '$' will be used as well, but inside of quotation marks and curly braces. The first element counts as zero. So if you want to print the first element of the array, you will be using '***[0]***'. If you looked at the example carefully, you would see there are '***[*]***' and '***[@]***'. They both mean to print all of the elements inside the array. Let's take a look at the output of the script above.

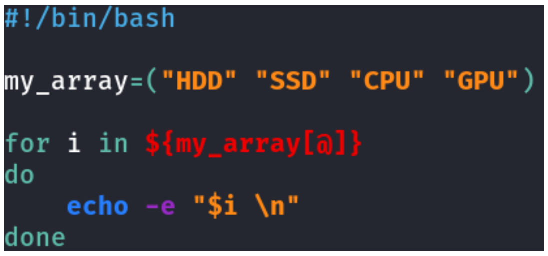


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_scripting_array.sh
second_element
Dodge
Asus Lenovo Dell
Kali Ubuntu Arch Mint
```

As you can see, the '***[1]***' printed the second element inside of the array. The '***[-1]***' printed the last element inside of the array. The '***[*]***' and '***[@]***' printed all of the elements inside of the array.

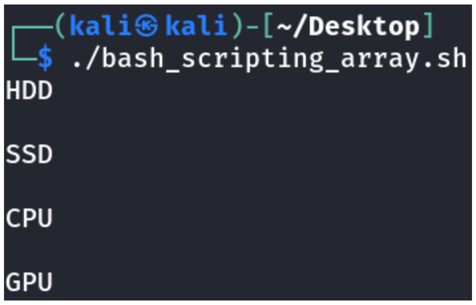
Iterating Elements Over Array

To perform iteration operations over an array, we can use loops. Let's see it as an example.



```
#!/bin/bash

my_array=("HDD" "SSD" "CPU" "GPU")

for i in ${my_array[@]}
do
    echo -e "$i \n"
done
```

As you can see, we have used '*for*' loop to print the elements of the array. The '***${my_array[@]}***' expands into all the elements in the array and the '*for*' loop iterates over them one by one with the iterator, in this case it's '**i**', but we can use something different.

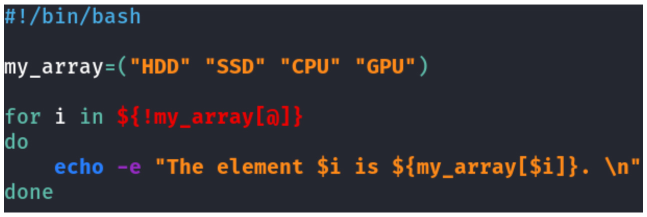


```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_scripting_array.sh
HDD

SSD

CPU

GPU
```

As you can see, the '*for*' loop printed all of the elements of '***my_array***'.

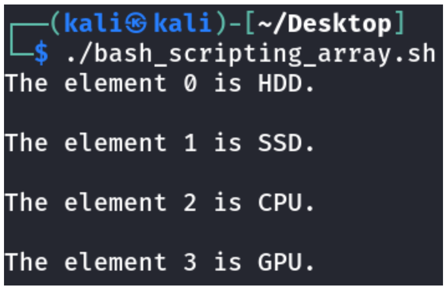
Printing Array Indices

To print the indices of the array, we can use loops again. It will be useful when we want to print the index of each element. Let's create a script.



```
#!/bin/bash

my_array=("HDD" "SSD" "CPU" "GPU")

for i in ${!my_array[@]}
do
    echo -e "The element $i is ${my_array[$i]}. \n"
done
```

As you can see, we used '*for*' loop again. Maybe you could see that there is an exclamation mark (!) at the beginning of '*${!my_array[@]}*'. This indicates that we are accessing the indices of the array and not the elements of the array themselves.



```
┌──(kali㉿kali)-[~/Desktop]
└─$ ./bash_scripting_array.sh
The element 0 is HDD.

The element 1 is SSD.

The element 2 is CPU.

The element 3 is GPU.
```

The '*for*' loop returned the index number of each element by order. If you want your output to look better, start counting from one, you can assign a new variable to one and increase it on each loop.

Things to Pay Attention in Bash Scripting

1. The Shebang line is important. If Shebang line wasn't specified in the first line of the script file, the interpreter used depends on the OS (Operating System) implementation and current shell. However, there is a way to run the Bash script file even though the Shebang line is missed. The Bash script file can be run by calling the script as an argument. For instance, '***bash my_bash_script_file***'.
2. Don't name the Bash script file as '***test***'. That's because the executable '***test***' already exists. The '***test.sh***' file would work anyway, but it's better to avoid it.
3. While setting the variables inside of a Bash script file, don't put a Dollar ($) sign before the variable name. For example, '***$my_var=5***' is wrong, '***my_var=5***' is correct. Use it when calling the variable itself.
4. Don't put a space on either or both sides of the equal sign (=) while assigning a value to variable. Otherwise, it will fail. For example, '***my_var = 5***' , '***my_var= 5'*** and '***my_var =5***' will all fail while '***my_var=5***' work.
5. There is an alternative way to react the exit code without using it. Let's say you want your script to do specific tasks by using if-else statement based on the result of '***$?***' (exit status code). Instead of specifying like '**if [$? -ne 0]**', you can use '**if ! cd not_exist**'. The exclamation mark (!) represents '*not equal.*', so the if block will be executed only if the command fails. That brings us to same point as using '***exit status code***', but simplified.
6. It's important to remember the different ways to run a child command, and whether you want the output, the return value or neither. When you want to run a command and ***save the output***, whether as a string or an array, you use Bash's '***$(command)***' syntax. For instance, '***my_var=$(printf "Hello world!")***'. When you want to use the ***return value*** of a command, just use the command, or add () to run a command or pipeline in subshell. For example, '**if (w | grep user | grep nmap)**'.
7. Bash does not typically require curly braces ({}) for variables, but it does for arrays. So you will notice that when you reference an array, you do so with the syntax '***${array_name}***', but when you reference a string or number, you simply use a Dollar sign '***$variable_name***'

Sources

https://www.youtube.com/watch?v=2733cRPudvI&list=PLT98CRl2KxKGj-VKtApD8-zCqSaN2mD4w&pp=iAQB

https://linuxhandbook.com/shebang/

https://www.geeksforgeeks.org/variables-programming/

https://phoenixnap.com/kb/bash-math

https://linuxhandbook.com/if-else-bash/

https://linuxize.com/post/bash-exit/

https://www.geeksforgeeks.org/bash-scripting-while-loop/

https://www.geeksforgeeks.org/bash-scripting-for-loop/

https://en.wikipedia.org/wiki/Standard_streams

https://en.wikipedia.org/wiki/Standard_streams

https://ryanstutorials.net/bash-scripting-tutorial/bash-functions.php

https://www.geeksforgeeks.org/bash-scripting-functions/

https://www.geeksforgeeks.org/bash-scripting-case-statement/

https://linuxize.com/post/bash-case-statement/

https://linuxhandbook.com/bash-until-loop/

https://www.geeksforgeeks.org/bash-scripting-array/

https://www.freecodecamp.org/news/bash-array-how-to-declare-an-array-of-strings-in-a-bash-script/

https://web.archive.org/web/20230330234404/https://wiki.bash-hackers.org/scripting/newbie_traps