**Experiment No.1**

Introduction to bash and shell scripting

• Setting Up Bash, configurations

• Shell Scripts for various logical and arithmetic tasks

• Shell scripts for various system tasks

**Topics to be covered:**

Introduction to Bash and Shell scripting

1. Input

2. Output

3. Streams

3.1 Stdin

3.2 Stdout

3.3 Stderr

4. Variable

5. Substitution

5.1 Variable Substitution

5.2 Command Substitution

6. Function

7. Sub-Shell

8. Conditional statements

8.1 IF statement Nested IF Ladder IF

8.2 CASE statement

9. Loops

9.1 FOR loop

9.2 WHILE loop

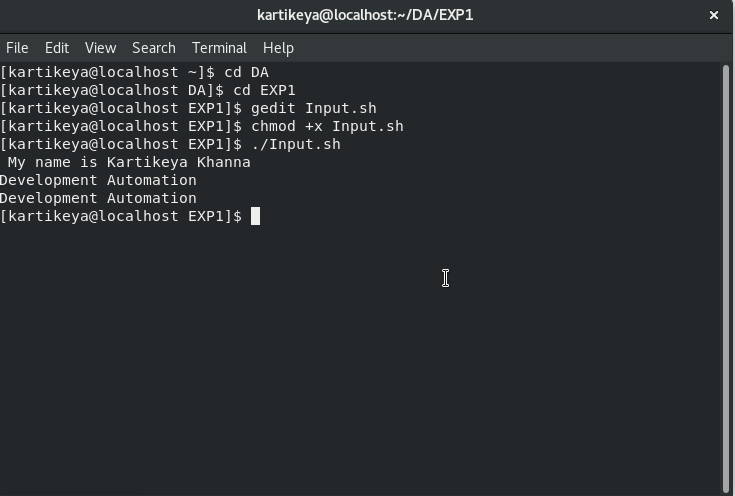
**1. Input**

A script can make use of the Input given to the script to Improve the user accessibility and also it enables the script to get data dynamically during the runtime.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | echo " My name is Kartikeya Khanna" |
|  | read course |
|  | echo "${course}" |

**Output**

****

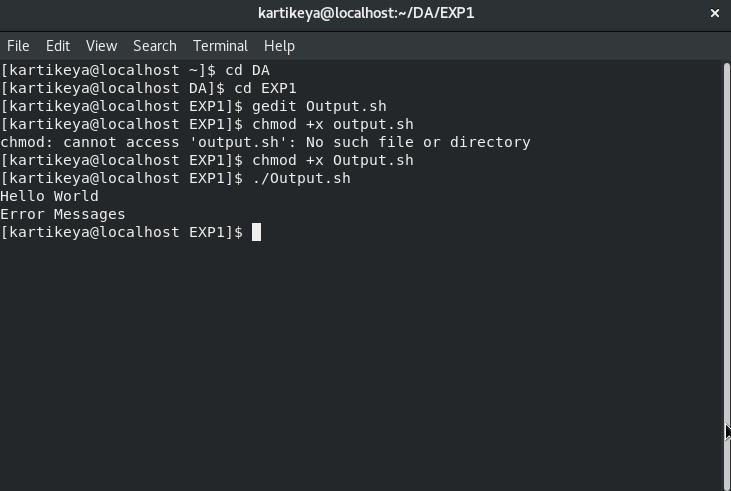
|  |
| --- |
|  |

**2. Output**

In Linux, the output is given to the terminal as two streams namely, “stdout” and “stderr”. The former one is used to print any general information to the terminal while the latter one is used to display the error information in the terminal.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | echo "Hello World" |
|  | echo "Error Messages">&2 |

**Output**

**3. Streams**

Streams provide an abstraction of communication channels between a computer program and its execution environment. Primarily, there are 3 kinds of streams in most of the Linux operating systems, namely:

1. Stdin
2. Stdout
3. Stderr

1.Stdin

Standard Input is a stream of data that enters the program from the terminal. Unless any output is redirected to the program, the standard input is expected from the keyboard. This stream is generally used in programs to read the data dynamically.

##### 2.Stdout

##### Standard Output is the stream that is used to display an information on the terminal. By default, all the text printed in the terminal will be sent to stdout. This can be a help message, a warning or an information that makes the program easier to use and know the process going on.

##### 3.Stderr

##### Standard Error is a specific stream used to display the error information to the terminal. It is separated from stdout so that the error messages can be captured separately. With this separation, the general output of a program can be shown directly on the terminal while the error-related information can be separately written to a file.

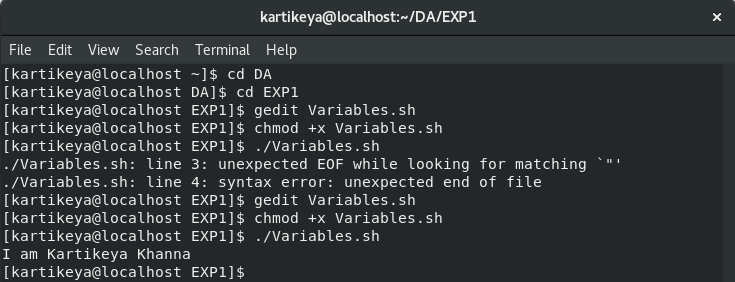
**4.Variable**

A variable is a space labelled by a name in memory that has some data inside. In other words, a variable is a labelled container that contains some data. The data of a variable can be changed wherever requires unless it is declared as read-only.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | myname="Kartikeya Khanna" |
|  | echo "I am ${myname}" |

**Output**

****

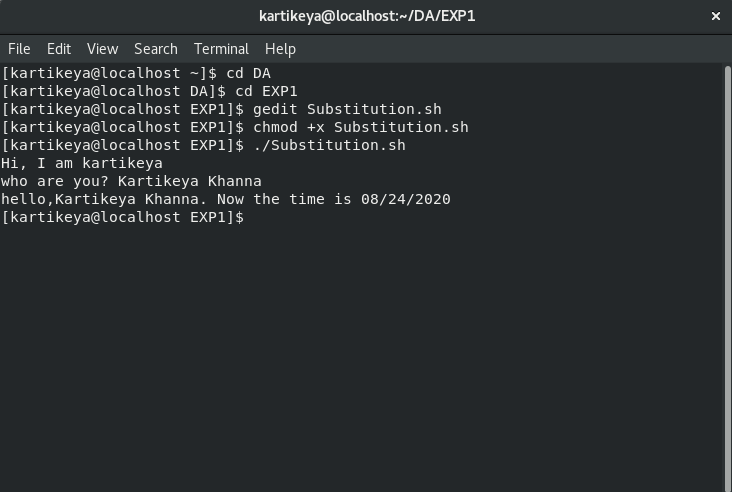
**5.Substitution**

Substitution generally refers to placing a text inside some other text. At a higher level, any command or a script is just a text in a particular format. This enables to use the constructed text in place of any commands.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | readonly greeting="hello" |
|  | current\_time=`date +%x` |
|  | echo "Hi, I am $(whoami)" |
|  | read -e -p "who are you?" myname |
|  | echo "${greeting},$myname. Now the time is $current\_time" |

**Output**

****

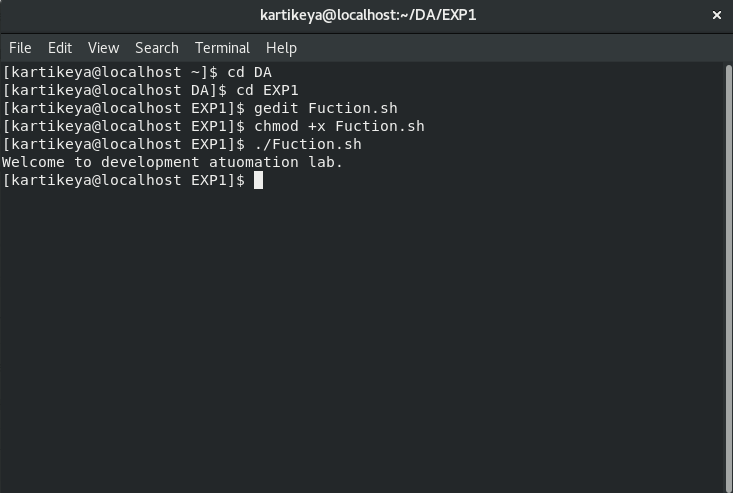
**6.Function**

Functions are used to define a set of actions and execute those actions multiple times across the program whenever they are called. Code duplication can be considerably reduced with the use of functions.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | function welcome { |
|  | echo "Welcome to $1" |
|  | } |
|  | welcome "development atuomation lab." |

**Output**

****

### **7.Sub-Shell**

### The shell is the terminal where we type the commands and execute the programs. This interactive shell can itself call a script that runs in a new process other than the parent shell’s process. Likewise, a running script can run some other script in a new process and this is called as a sub-shell.

### **8.Conditional statements**

### Conditional statements are the vital components in any scripting or programming languages. They enable the ability to take decisions based on the given Boolean values. This allows us to logically do actions based on some criteria.

Much like regular programming languages, Bash supports two types of conditional statements:

1.IF statements

2.CASE statements

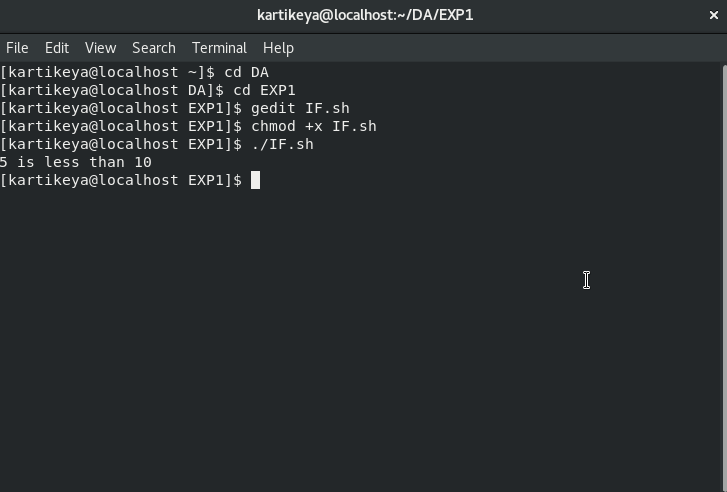
#### 1.IF statement

The IF statement takes up a Boolean value directly and executes the given code if the given condition is true. Expressions that gets evaluated to Boolean can be given as inputs to the IF statements.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | num1=5 |
|  | num2=10 |
|  | if [ $num1 -le $num2 ]; then |
|  | echo "$num1 is less than $num2" |
|  | else |
|  | echo "$num1 is greater than $num2" |
|  | fi |

**Output**

****

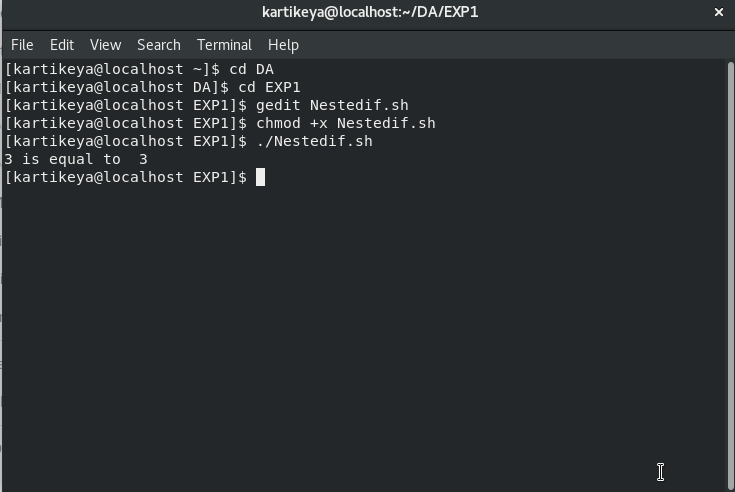
Nested IF

An IF statement can have multiple IF statements in its action and this type is generally called as “Nested IF” statement. If the actions don’t have any commands other than another IF statement, then the evaluation can be shrunk at one place rather than having a Nested IF statement.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | num1=3 |
|  | num2=3 |
|  | if [ $num1 -ne $num2 ]; then |
|  | if [ $num1 -gt $num2]; then |
|  | echo "$num1 is greater than $num2" |
|  | else |
|  | echo "$num1 is lesser than $num2" |
|  | fi |
|  | else |
|  | echo "$num1 is equal to $num2" |
|  | fi |

**Output**

****

Ladder IF

In some scenarios, multiple conditions need to be checked one after another till a condition that resolves to true is met. Instead of having multiple IF statements inside the “else” clause of each parent IF statement, multiple **“elif”** keyword with conditions and code for each can be used.

**Code**

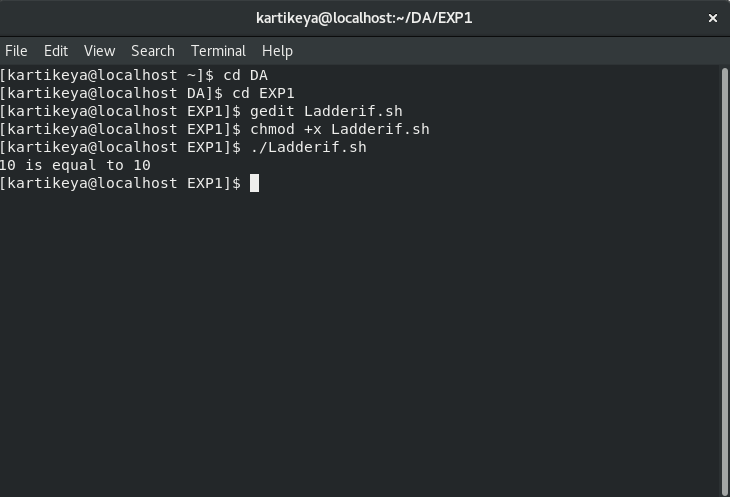
Top of Form

Bottom of Form

Top of Form

Bottom of Form

|  |  |
| --- | --- |
|  | #!/bin/bash |
|  | num1=10 |
|  | num2=10 |
|  | if [ $num1 -gt $num2 ]; then |
|  | echo "$num1 is greater then $num2" |
|  | elif [ $num1 -lt $num2 ]; then |
|  | echo "$num1 is lesser than $num2" |
|  | else |
|  | echo "$num1 is equal to $num2" |
|  | fi |

**Output****

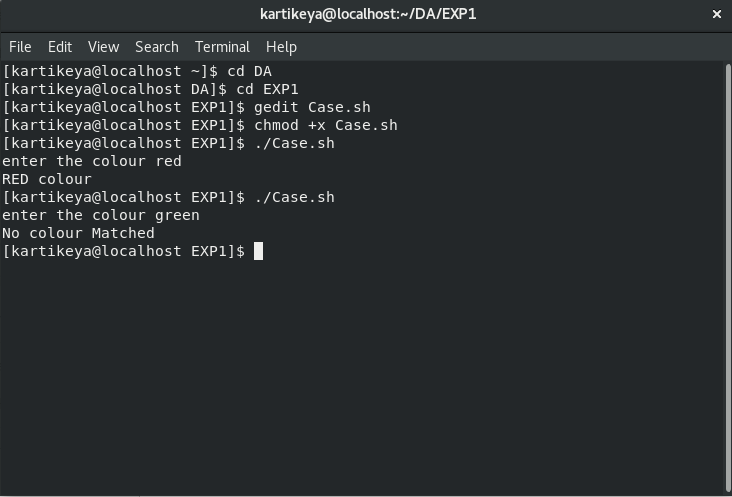
*2.*CASE statement

It is also a conditional statement that allows controlling the execution flow of a program. It is almost equivalent to the ladder IF statement. But the syntax and way of providing conditions are slightly different than the ladder IF statement.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | read -e -p "enter the colour " colour |
|  | case "$colour" in |
|  | red) echo "RED colour";; |
|  | black) echo "Black colour";; |
|  | \*) echo "No colour Matched";; |
|  | esac |

**Output**

****

### **9.Loops**

### Loops are also a kind of control flow statement that enables the program to iterate a defined piece of code for a particular number of times or till a specified condition gets false. Loops are of two types:

1. For loop
2. While loop

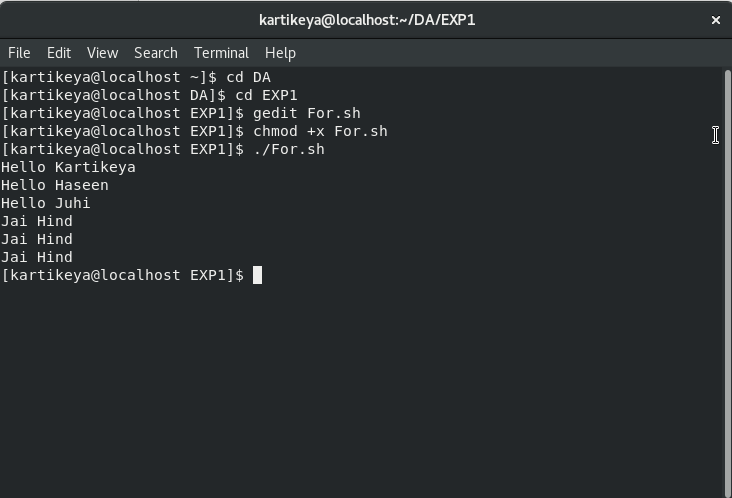
1.For loop

FOR loop is a common type of looping statement used to iterate over a given number of items. Usually, an array of items will be given as input to FOR statement. This statement starts with the first item and switches to the next item in each iteration till it goes through all of them in the given array.

**Code**

|  |
| --- |
| #!/bin/bash |
|  | names="Kartikeya Haseen Juhi" |
|  | for name in $names; do |
|  | echo "Hello $name" |
|  | done |
|  | for number in seq 1 5; do |
|  | echo "Jai Hind" |
|  | done |

**Output**

****

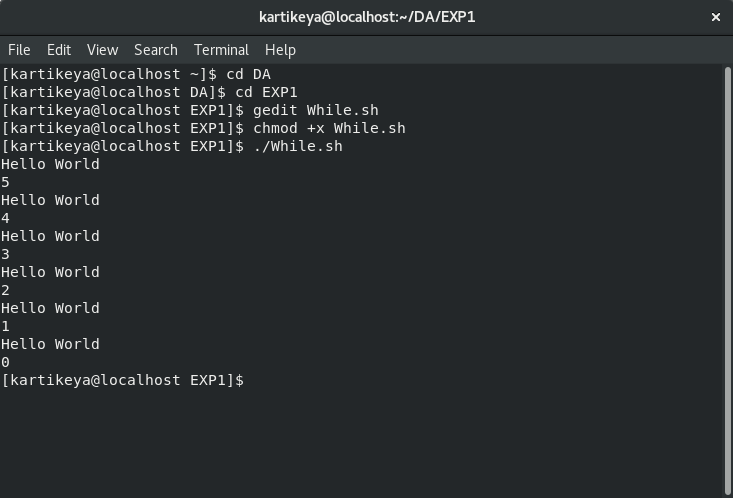
#### 2.While loop

#### WHILE loop is also a statement to do iterations same like FOR loop. The only difference is that WHILE loop does iteration till a given condition becomes false whereas the FOR statement does iteration over the given number of items.

**Code**

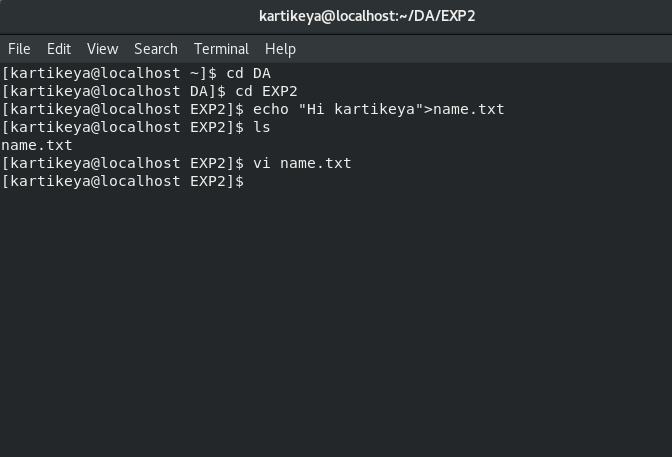
|  |
| --- |
| #!/bin/bash |
|  | names="Kartikeya Haseen Juhi" |
|  | for name in $names; do |
|  | echo "Hello $name" |
|  | done |
|  | for number in seq 1 5; do |
|  | echo "Jai Hind" |
|  | done |

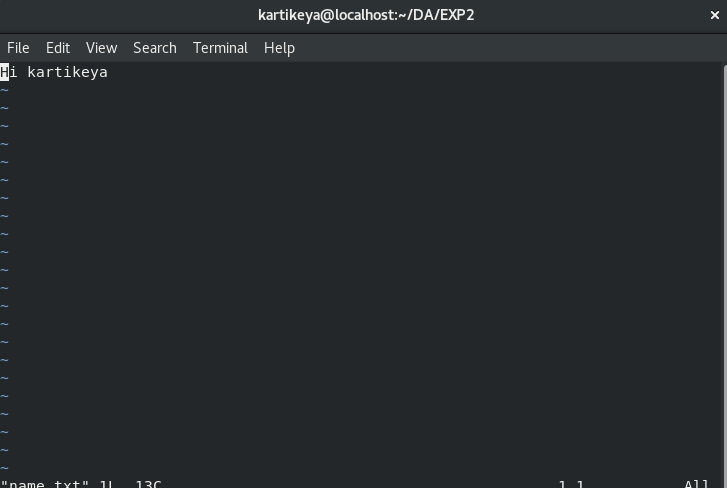
**Output**

****

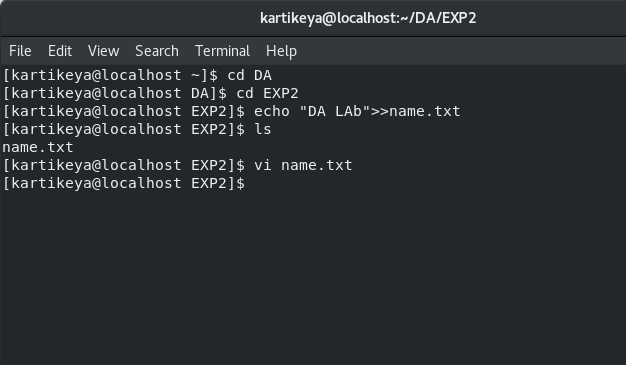
**Experiment No.2**

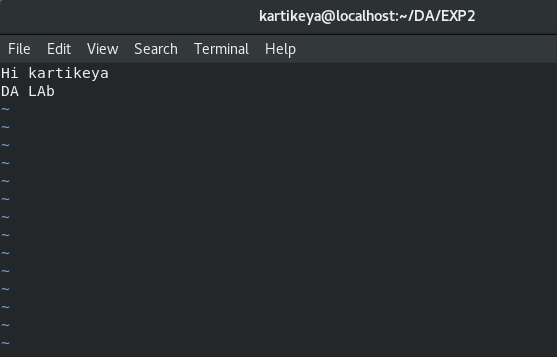
**1.Standard Input**

****

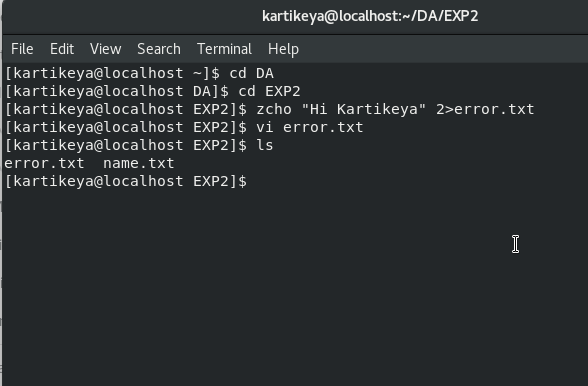
****

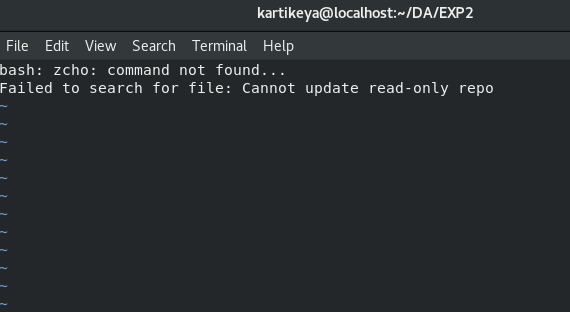
**2.Append**

****

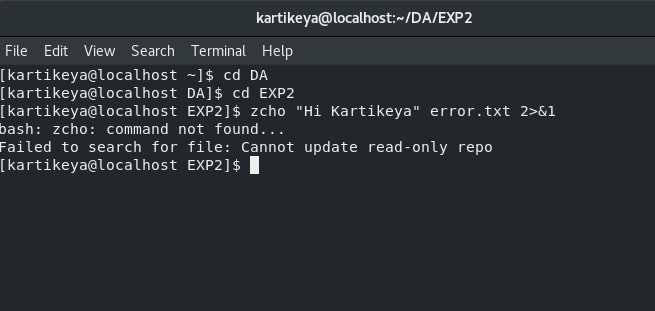
****

**3.Errors**

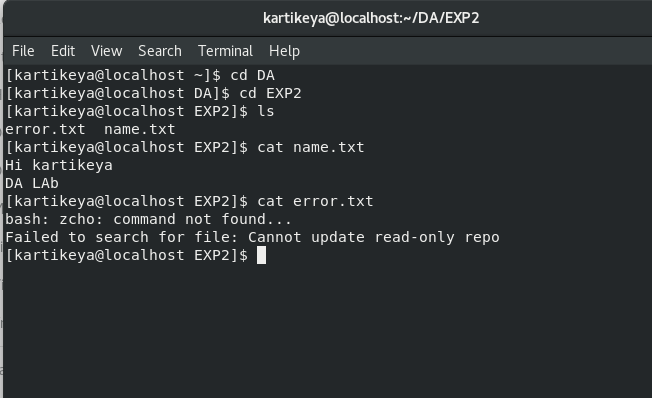
****

****

**4.Error Redirection**

****

**5.Cat Command**

****

**Experiment No.3**

###### **Aim:**

To write a script that provides a set of functions to format the terminal as listed below:

1.Colorize the given text

2.Show progress bar that fills over a specified time

3.Transform the case of the given text

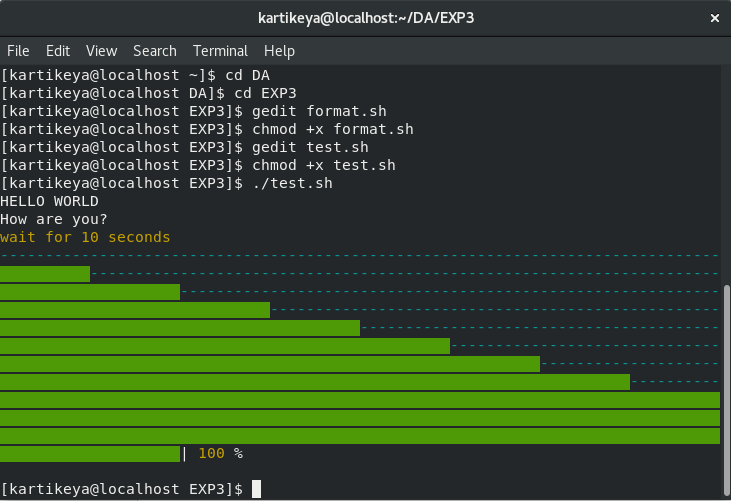
**Codes:**

**1.Format.sh**

|  |
| --- |
| #!/bin/bash |
|  | style() |
|  | { |
|  | echo "\033[$1m" |
|  | } |
|  | red\_text() |
|  | { |
|  | printf "$(style "31")$1$(style "0")"; |
|  | } |
|  | green\_text() |
|  | { |
|  | printf "$(style "32")$1$(style "0")"; |
|  | } |
|  | yellow\_text() |
|  | { |
|  | printf "$(style "33")$1$(style "0")"; |
|  | } |
|  | blue\_text() |
|  | { |
|  | printf "$(style "34")$1$(style "0")"; |
|  | } |
|  | cyan\_text() |
|  | { |
|  | printf "$(style "36")$1$(style "0")"; |
|  | } |
|  | progress\_bar() |
|  | { |
|  | if [ -z "$1" ] ; then |
|  | echo "ERROR: The duration of the progress bar is required" >&2; exit 1; |
|  | Fi |
|  | local duration=$1 |
|  | local increment=$((100/$duration)) |
|  | for (( elapsed=0; elapsed<=100; elapsed=elapsed+increment )); do for ((done=0; done<elapsed; done=done+1)); do |
|  | printf "$(green\_text "▇")"; done |
|  | for ((remain=elapsed; remain<100; remain=remain+1)); do printf "$(cyan\_text "-")"; |
|  | Done |
|  | printf "| $(yellow\_text "$elapsed") %%" ; sleep 1 |
|  | printf "\r"; done |
|  | echo -e "\n"; |
|  | } |
|  |  |
|  | camelcase() |
|  | { |
|  | echo $1 | awk '{print toupper(substr($0,0,1))tolower(substr($0,2))}' |
|  | } |
|  |  |
|  | uppercase() |
|  | { |
|  | echo $1 | awk '{print toupper($0)}' |
|  | } |
|  | lowercase() |
|  | { |
|  | echo $1 | awk '{print tolower($0)}' |
|  | } |

**2.Test.sh**

|  |
| --- |
| #!/bin/bash |
|  | . /home/kartikeya/DA/EXP3/format.sh |
|  | uppercase "hello world" |
|  | camelcase "how are you?" |
|  | yellow\_text "wait for 10 seconds" |
|  | echo " " |
|  | progress\_bar 10 |

**Output**

**Experiment No.4**

###### **Aim:**

To write a script that provides a list of utilities to reuse across the scripts. The utilities are listed below:

1.Check whether the given user exists

2.Check whether the root permission is given

3.Exit the function with an error message

4.Check whether a given path is valid and is a file

**Codes:**

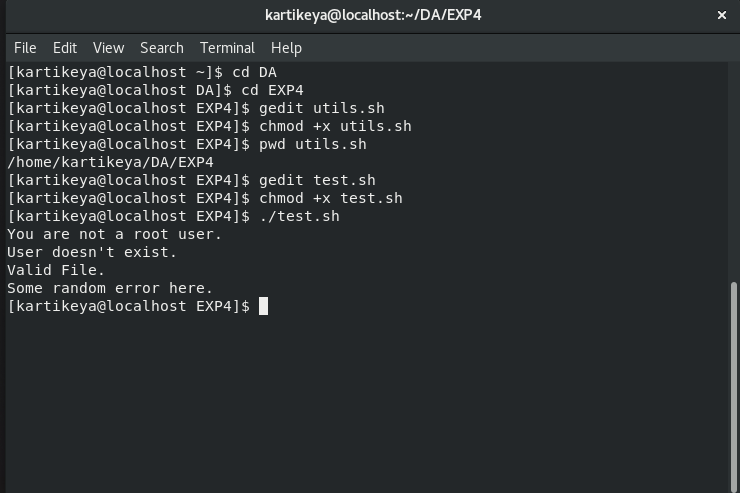
**1.Utils.sh**

|  |
| --- |
| #!/bin/bash |
|  | declare -r TRUE=0 |
|  | declare -r FALSE=1 |
|  | die() |
|  | { |
|  | echo "$1">&2; |
|  | exit 1 |
|  | } |
|  | has\_root\_permission() |
|  | { |
|  | [ $(id -u) -eq 0 ] && return $TRUE || return $FALSE |
|  | } |
|  | does\_user\_exit() |
|  | { |
|  | local username="$1" |
|  | grep -q "^${username}" /etc/passwd && return $TRUE || return $FALSE |
|  | } |
|  | is\_valid\_file() |
|  | { |
|  | local given\_path="$1" |
|  | [ -f "$(realpath $given\_path)" ] && return $TRUE || return $FALSE |
|  | } |

**2.Test.sh**

|  |
| --- |
| #!/bin/bash |
|  | . /home/kartikeya/DA/EXP4/utils.sh |
|  | has\_root\_permission && echo "You are a root user." || echo "You are not a root user." |
|  | does\_user\_exit "Redhat" && echo "User exist." || echo "User doesn't exist." |
|  | is\_valid\_file "/etc/passwd" && echo "Valid File." || echo "Invalid File." |
|  | die "Some random error here." |

**Output**

****

**SUMBITED BY: KARTIKEYA NARIAN KHANNA**

**Roll No: 049**

**Course: B.Tech CSE Devops.**