



Shell Script Examples

Last Updated : 28 Mar, 2024

For all the Linux distributions, the shell script is like a magic wand that automates the process, saves users time, and increases productivity. This shell scripting tutorial will introduce you to the 25 plus shell scripting examples.

But before we move on to the topic of [shell scripting](#) examples, let's understand shell script and the actual use cases of shell scripting.

What is Shell Script?

Well, the shell is a **CLI (command line interpreter)**, which runs in a text window where users can manage and execute shell commands. On the other hand, the process of writing a set of commands to be executed on a Linux system. A file that includes such instructions is called a bash script.

Uses of Shell Scripts

Below are some common uses of Shell Script:

- **Task Automation** - It can be used to automate repetitive tasks like regular backups and software installation tasks.
- **Customization** - One can use shell scripts to design its command line environment and easily perform its task as per needs.
- **File Management** - The shell scripts can also be used to manage and manipulate files and directories, such as moving, copying, renaming, or deleting files.

Shell Script Examples in Linux

1) What does the shebang (#!) at the beginning of a shell script indicate?

The shebang (#!) at the beginning of a script indicates the interpreter that should be used to execute the script. It tells the system which shell or interpreter should interpret the script's commands.

For example: Suppose we have a script named **myscript.sh** written in the Bash shell:

```
#!/bin/bash

echo "This script is using the Bash interpreter."
echo "Hello, GeeksforGeeks!"
```

shebang

In this example:

- The `#!/bin/bash` at the beginning of the script indicates that the script should be interpreted using the Bash shell.
- The `echo` commands are used to print messages to the terminal.

2) How do you run a shell script from the command line?

To run a shell script from the command line, we need to follow these steps:

- Make sure the script file has executable permissions using the [chmod command](#):

```
chmod +x myscript.sh
```

- Execute the script using its filename:

```
./myscript.sh
```

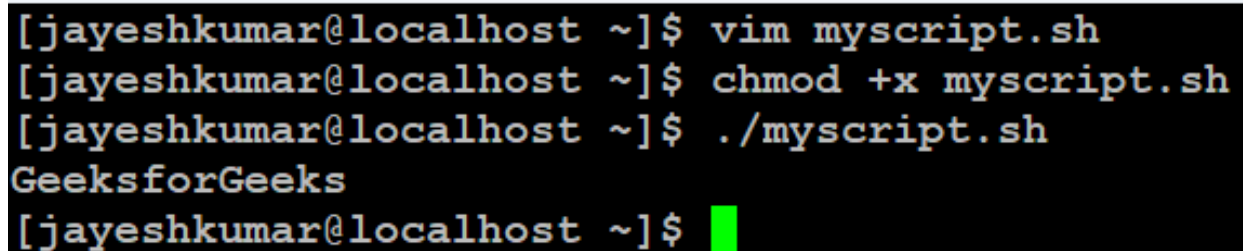
Here you have to replace "**myscript.sh**" with yors script name.

3) Write a shell script that prints "GeeksforGeeks" to the terminal.

Create a script name ``myscript.sh`` (we are using ``vim`` editor, you can choose any editor)

```
vim myscript.sh
```

```
#!/bin/bash  
# This script prints "GeeksforGeeks" to the terminal  
echo "GeeksforGeeks"
```



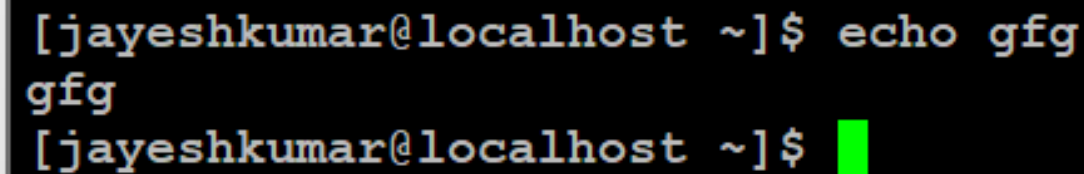
```
[jayeshkumar@localhost ~]$ vim myscript.sh  
[jayeshkumar@localhost ~]$ chmod +x myscript.sh  
[jayeshkumar@localhost ~]$ ./myscript.sh  
GeeksforGeeks  
[jayeshkumar@localhost ~]$
```

print name

We make our script executable by using ``chmod +x`` then execute with ``./myscript.sh`` and get our desired output "GeeksforGeeks".

4) Explain the purpose of the echo command in shell scripting.

The [echo command](#) is used to display text or variables on the terminal. It's commonly used for printing messages, variable values, and generating program output.



```
[jayeshkumar@localhost ~]$ echo gfg  
gfg  
[jayeshkumar@localhost ~]$
```

echo command

In this example we have execute `echo` on terminal directly , as it works same inside shell script.

5) How can you assign a value to a variable in a shell script?

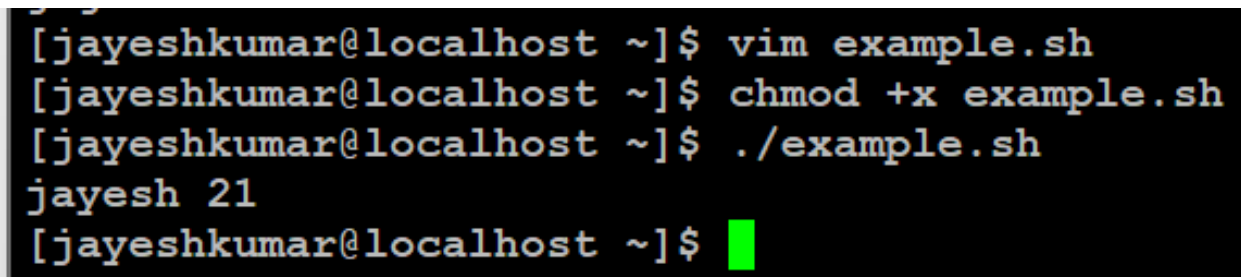
Variables are assigned values using the assignment operator =.

For example:

```
#!/bin/bash
# Assigning a value to a variable
name="Jayesh"
age=21
echo $name $age
```

Explanation:

- The name variable is assigned the value "Jayesh".
- The age variable is assigned the value 21.
- echo is used to print and ` \$name ` ` \$age ` is used to call the value stored in the variables.

A terminal window with a black background and green text. The prompt is [jayeshkumar@localhost ~]. The user enters 'vim example.sh', then 'chmod +x example.sh', and finally './example.sh'. The output of the script is 'jayesh 21'. The prompt returns to [jayeshkumar@localhost ~].

```
[jayeshkumar@localhost ~]$ vim example.sh
[jayeshkumar@localhost ~]$ chmod +x example.sh
[jayeshkumar@localhost ~]$ ./example.sh
jayesh 21
[jayeshkumar@localhost ~]$
```

6) Write a shell script that takes a user's name as input and greets them.

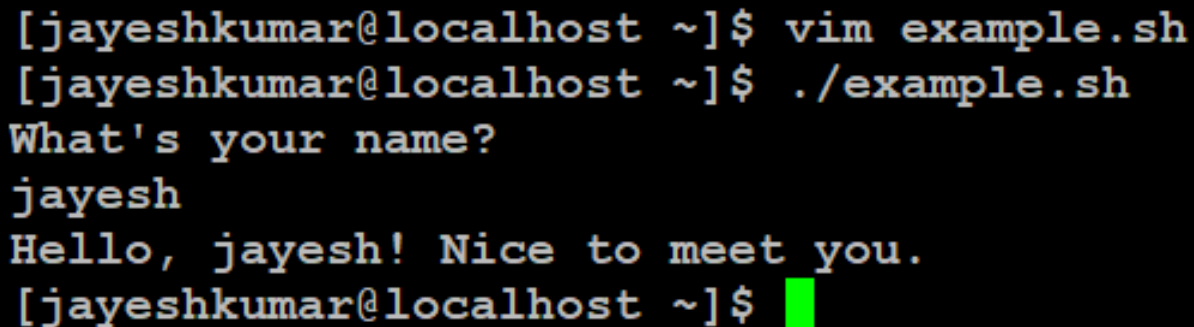
Create a script name `example.sh`.

```
#!/bin/bash
# Ask the user for their name
echo "What's your name?"
read name
```

```
# Greet the user  
echo "Hello, $name! Nice to meet you."
```

Explanation:

- `#!/bin/bash`: This is the shebang line. It tells the system to use the Bash interpreter to execute the script.
- `# Ask the user for their name`: This is a comment. It provides context about the upcoming code. Comments are ignored by the interpreter.
- `echo "What's your name?"`: The echo command is used to display the text in double quotes on the terminal.
- `read name`: The read command waits for the user to input text and stores it in the variable name.
- `echo "Hello, $name! Nice to meet you."`: This line uses the echo command to print a greeting message that includes the value of the name variable, which was collected from the user's input.



```
[jayeshkumar@localhost ~]$ vim example.sh  
[jayeshkumar@localhost ~]$ ./example.sh  
What's your name?  
jayesh  
Hello, jayesh! Nice to meet you.  
[jayeshkumar@localhost ~]$
```

7) How do you add comments to a shell script?

[Comments in shell scripting](#) are used to provide explanations or context to the code. They are ignored by the interpreter and are only meant for humans reading the script. You can add comments using the `#` symbol.

```
#!/bin/bash  
# This is a comment explaining the purpose of the script  
echo "gfg"
```

8) Create a shell script that checks if a file exists in the current directory.

Here's a script that checks if a file named "example.txt" exists in the current directory:

```
#!/bin/bash
file="example.txt"
# Check if the file exists
if [ -e "$file" ]; then
echo "File exists: $file"
else
echo "File not found: $file"
fi
```

Explanation:

1. `#!/bin/bash`: This is the shebang line that specifies the interpreter (`/bin/bash`) to be used for running the script.
2. `file="example.txt"`: This line defines the variable `file` and assigns the value "example.txt" to it. You can replace this with the name of the file you want to check for.
3. `if [-e "$file"]; then`: This line starts an if statement. The condition `[-e "$file"]` checks if the file specified by the value of the `file` variable exists. The `-e` flag is used to check for file existence.
4. `echo "File exists: $file"`: If the condition is true (i.e., the file exists), this line prints a message indicating that the file exists, along with the file's name.
5. `else`: If the condition is false (i.e., the file doesn't exist), the script executes the code under the else branch.
6. `echo "File not found: $file"`: This line prints an error message indicating that the specified file was not found, along with the file's name.

7. fi: This line marks the end of the if statement.

```
[jayeshkumar@localhost ~]$ touch example.txt
[jayeshkumar@localhost ~]$ vim find.sh
[jayeshkumar@localhost ~]$ chmod +x find.sh
[jayeshkumar@localhost ~]$ ./find.sh
File exists: example.txt
[jayeshkumar@localhost ~]$
```

Finding file

9) What is the difference between single quotes (') and double quotes (") in shell scripting?

Single quotes (') and double quotes (") are used to enclose strings in shell scripting, but they have different behaviors:

- Single quotes: Everything between single quotes is treated as a literal string. Variable names and most special characters are not expanded.
- Double quotes: Variables and certain special characters within double quotes are expanded. The contents are subject to variable substitution and command substitution.

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

```
abcd="Hello"
echo '$abcd' # Output: $abcd
echo "$abcd" # Output: Hello
```

10) How can you use command-line arguments in a shell script?

Command-line arguments are values provided to a script when it's executed. They can be accessed within the script using special variables

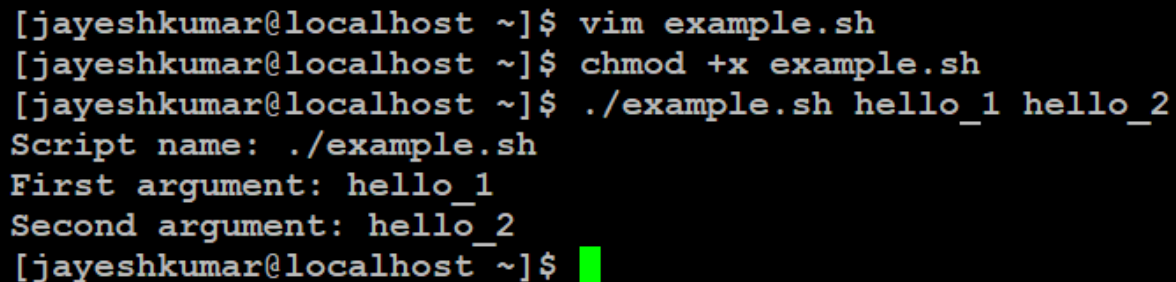
like \$1, \$2, etc., where \$1 represents the first argument, \$2 represents the second argument, and so on.

For Example: If our script name is `example.sh`

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

```
echo "Script name: $0"  
echo "First argument: $1"  
echo "Second argument: $2"
```

If we run the script with `./example.sh hello_1 hello_2`, it will output:

A terminal window with a black background and yellow text. The prompt is [jayeshkumar@localhost ~]\$. The user enters vim example.sh, then chmod +x example.sh, and finally ./example.sh hello_1 hello_2. The script outputs: Script name: ./example.sh, First argument: hello_1, and Second argument: hello_2. The prompt returns to [jayeshkumar@localhost ~]\$.

```
[jayeshkumar@localhost ~]$ vim example.sh  
[jayeshkumar@localhost ~]$ chmod +x example.sh  
[jayeshkumar@localhost ~]$ ./example.sh hello_1 hello_2  
Script name: ./example.sh  
First argument: hello_1  
Second argument: hello_2  
[jayeshkumar@localhost ~]$
```

cli arguments

11) How do you use the for loop to iterate through a list of values?

Create a script name `example.sh`.

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

```
fruits=("apple" "banana" "cherry" "date")  
for fruit in "${fruits[@]}"; do  
    echo "Current fruit: $fruit"  
done
```

Explanation:

``fruits=`` line creates an array named `fruits` with four elements: "apple", "banana", "cherry", and "date".

- `for fruit in "${fruits[@]}"; do`: This line starts a for loop. Here's what each part means:
- `for fruit`: This declares a loop variable called `fruit`. In each iteration of the loop, `fruit` will hold the value of the current element from the `fruits` array.
- `"${fruits[@]}"`: This is an array expansion that takes all the elements from the `fruits` array. The `"${...}"` syntax ensures that each element is treated as a separate item.
- `do`: This keyword marks the beginning of the loop body.
- `echo "Current fruit: $fruit"`: Inside the loop, this line uses the `echo` command to display the current value of the loop variable `fruit`. It prints a message like "Current fruit: apple" for each fruit in the array.
- `done`: This keyword marks the end of the loop body. It tells the script that the loop has finished.

```
[jayeshkumar@localhost ~]$ vim example.sh
[jayeshkumar@localhost ~]$ chmod +x example.sh
[jayeshkumar@localhost ~]$ ./example.sh
Current fruit: apple
Current fruit: banana
Current fruit: cherry
Current fruit: date
[jayeshkumar@localhost ~]$
```

for loop

12) Write a shell script that calculates the sum of integers from 1 to N using a loop.

Create a script name ``example.sh``.

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

```
echo "Enter a number (N):"  
read N  
sum=0  
for (( i=1; i<=$N; i++ )); do  
sum=$((sum + i))  
done  
echo "Sum of integers from 1 to $N is: $sum"
```

Explanation:

The script starts by asking you to enter a number (N) using read. This number will determine how many times the loop runs.

1. The variable sum is initialized to 0. This variable will keep track of the sum of integers.
2. The for loop begins with for ((i=1; i<=\$N; i++)). This loop structure is used to repeat a set of actions a certain number of times, in this case, from 1 to the value of N.
3. Inside the loop, these things happen:
 - i=1 sets the loop variable i to 1 at the beginning of each iteration.
 - The loop condition i<=\$N checks if i is still less than or equal to the given number N.
 - If the condition is true, the loop body executes.
 - sum=\$((sum + i)) calculates the new value of sum by adding the current value of i to it. This adds up the integers from 1 to the current i value.
4. After each iteration, i++ increments the value of i by 1.
5. The loop continues running until the condition i<=\$N becomes false (when i becomes greater than N).
6. Once the loop finishes, the script displays the sum of the integers from 1 to the entered number N.

```
[jayeshkumar@localhost ~]$ vim example.sh
[jayeshkumar@localhost ~]$ chmod +x example.sh
[jayeshkumar@localhost ~]$ ./example.sh
Enter a number (N) :
3
Sum of integers from 1 to 3 is: 6
[jayeshkumar@localhost ~]$
```

13) Create a script that searches for a specific word in a file and counts its occurrences.

Create a script name `word_count.sh`

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

```
echo "Enter the word to search for:"
read target_word
echo "Enter the filename:"
read filename
count=$(grep -o -w "$target_word" "$filename" | wc -l)
echo "The word '$target_word' appears $count times in '$filename'."
```

Explanation:

- `echo "Enter the word to search for:"`: This line displays a message asking the user to enter a word they want to search for in a file.
- `read target_word`: This line reads the input provided by the user and stores it in a variable named `target_word`.
- `echo "Enter the filename:"`: This line displays a message asking the user to enter the name of the file they want to search in.
- `read filename`: This line reads the input provided by the user and stores it in a variable named `filename`.
- `count=$(grep -o -w "$target_word" "$filename" | wc -l)`: This line does the main work of the script. Let's break it down further:

- `grep -o -w "$target_word" "$filename"`: This part of the command searches for occurrences of the `target_word` in the specified filename. The options `-o` and `-w` ensure that only whole word matches are counted.
- `|`: This is a pipe, which takes the output of the previous command and sends it as input to the next command.
- `wc -l`: This part of the command uses the `wc` command to count the number of lines in the input. The option `-l` specifically counts the lines.
- The entire command calculates the count of occurrences of the `target_word` in the file and assigns that count to the variable `coun`

```
[jayeshkumar@localhost ~]$ vim word_count.sh
[jayeshkumar@localhost ~]$ vim geeks.txt
[jayeshkumar@localhost ~]$ chmod +x word_count.sh
[jayeshkumar@localhost ~]$ ./word_count.sh
Enter the word to search for:
hello
Enter the filename:
geeks.txt
The word 'hello' appears 4 times in 'geeks.txt'.
[jayeshkumar@localhost ~]$ cat geeks.txt
hello , hi, hello, hi , hello , hi , hey , hello there, hi
[jayeshkumar@localhost ~]$
```

14) Explain the differences between standard output (stdout) and standard error (stderr).

The main difference between standard output (stdout) and standard error (stderr) is as follows:

- **Standard Output (stdout):** This is the default output stream where a command's regular output goes. It's displayed on the terminal by default. You can redirect it to a file using `>`.
- **Standard Error (stderr):** This is the output stream for error messages and warnings. It's displayed on the terminal by default as well. You can redirect it to a file using `2>`.

15) Explain the concept of conditional statements in shell scripting.

[Conditional statements in shell scripting](#) allow us to make decisions and control the flow of our script based on certain conditions. They enable our script to execute different sets of commands depending on whether a particular condition is true or false. The primary conditional statements in shell scripting are the if statement, the elif statement (optional), and the else statement (optional).

Here's the basic structure of a conditional statement in shell scripting:

```
if [ condition ]; then  
# Commands to execute if the condition is true  
elif [ another_condition ]; then  
# Commands to execute if another_condition is true (optional)  
else  
# Commands to execute if none of the conditions are true (optional)  
fi
```

Explanation:

- `[condition]` = Command that evaluates the condition and returns a true (0) or false (non-zero) exit status.
- `then` = It is a keyword which indicates that the commands following it will be executed if the condition evaluates to true.
- `elif` = (short for "else if") It is a section that allows us to specify additional conditions to check.
- `else` = it is a section that contains commands that will be executed if none of the conditions are true.
- `fi` = It is a keyword that marks the end of the conditional block.

16) How do you read lines from a file within a shell script?

To [read lines](#) from a file within a shell script, we can use various methods, but one common approach is to use a while loop in combination with the

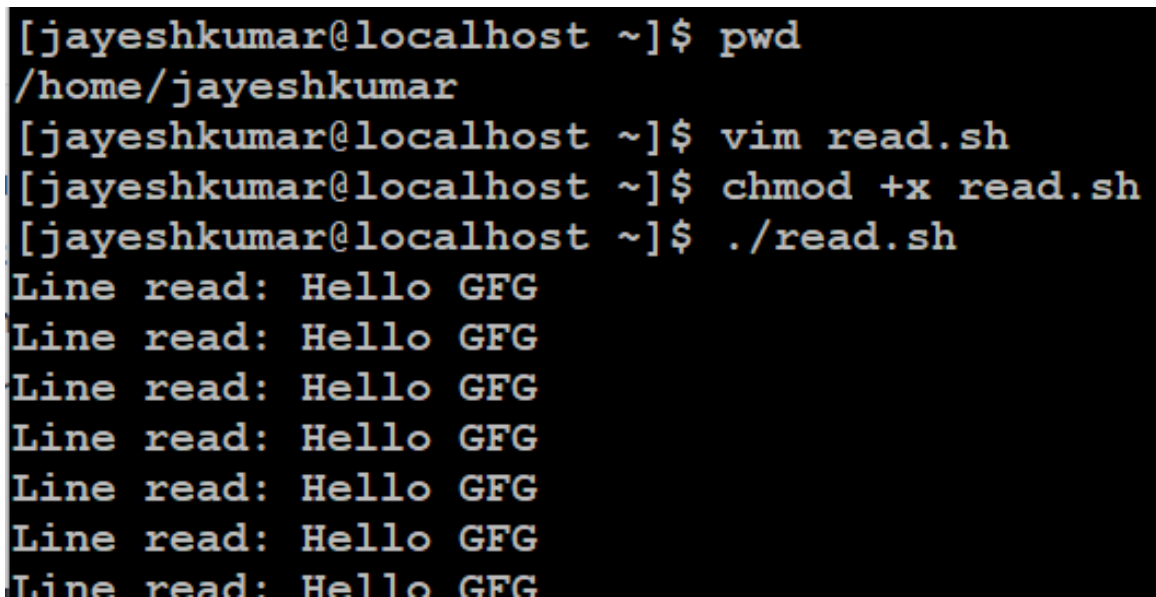
read command. Here's how we can do it:

```
#!/bin/bash  
file="/home/jayeshkumar/jayesh.txt"  
# Check if the file exists  
if [ -e "$file" ]; then  
while IFS= read -r line; do  
echo "Line read: $line"  
# Add your processing logic here  
done < "$file"  
else  
echo "File not found: $file"  
fi
```

Explanation:

1. `#!/bin/bash`: This is the shebang line that specifies the interpreter (`/bin/bash`) to be used for running the script.
2. `file="/home/jayeshkumar/jayesh.txt"`: This line defines the variable `file` and assigns the full path to the file `jayesh.txt` in the `/home/jayeshkumar` directory. Change this path to match the actual path of the file you want to read.
3. `if [-e "$file"]; then`: This line starts an if statement. It checks if the file specified by the variable `$file` exists. The `-e` flag checks for file existence.
4. `while IFS= read -r line; do`: This line initiates a while loop that reads lines from the file.
 - `IFS=`: The IFS (Internal Field Separator) is set to an empty value to preserve leading and trailing spaces.
 - `read -r line`: This reads the current line from the file and stores it in the variable `line`.
5. `echo "Line read: $line"`: This line prints the content of the line that was read from the file. The variable `$line` contains the current line's content.

6. # Add your processing logic here: This is a placeholder comment where you can add your own logic to process each line. For example, you might analyze the line, extract information, or perform specific actions based on the content.
7. done < "\$file": This marks the end of the while loop. The < "\$file" redirects the content of the file to be read by the loop.
8. else: If the file does not exist (the condition in the if statement is false), the script executes the code under the else branch.
9. echo "File not found: \$file": This line prints an error message indicating that the specified file was not found.
10. fi: This line marks the end of the if statement.

A terminal window with a black background and white text. The prompt is [jayeshkumar@localhost ~]\$. The user enters 'pwd' and the output is '/home/jayeshkumar'. Then the user enters 'vim read.sh'. Next, the user enters 'chmod +x read.sh'. Finally, the user enters './read.sh'. The script then prints 'Line read: Hello GFG' seven times.

```
[jayeshkumar@localhost ~]$ pwd
/home/jayeshkumar
[jayeshkumar@localhost ~]$ vim read.sh
[jayeshkumar@localhost ~]$ chmod +x read.sh
[jayeshkumar@localhost ~]$ ./read.sh
Line read: Hello GFG
Line read: Hello GFG
Line read: Hello GFG
Line read: Hello GFG
Line read: Hello GFG
Line read: Hello GFG
Line read: Hello GFG
```

reading file

Here , we used `pwd` command to get the path of current directory.

17) Write a function in a shell script that calculates the factorial of a given number.

Here is the script that calculate the factorial of a given number.

```
#!/bin/bash
# Define a function to calculate factorial
calculate_factorial() {
    num=$1
    fact=1
```

```
for ((i=1; i<=num; i++)); do
fact=$((fact * i))
done
echo $fact
}
# Prompt the user to enter a number
echo "Enter a number: "
read input_num
# Call the calculate_factorial function with the input number
factorial_result=$(calculate_factorial $input_num)
# Display the factorial result
echo "Factorial of $input_num is: $factorial_result"
```

Explanation:

1. The script starts with the shebang line `#!/bin/bash` to specify the interpreter.
2. `calculate_factorial()` is defined as a function. It takes one argument, `num`, which is the number for which the factorial needs to be calculated.
3. Inside the function, `fact` is initialized to 1. This variable will store the factorial result.
4. The `for` loop iterates from 1 to the given number (`num`). In each iteration, it multiplies the current value of `fact` by the loop index `i`.
5. After the loop completes, the `fact` variable contains the calculated factorial.
6. The script prompts the user to enter a number using `read`.
7. The `calculate_factorial` function is called with the user-provided number, and the result is stored in the variable `factorial_result`.
8. Finally, the script displays the calculated factorial result.


```
[jayeshkumar@localhost ~]$ vim factorial.sh
[jayeshkumar@localhost ~]$ chmod +x factorial.sh
[jayeshkumar@localhost ~]$ ./factorial.sh
Enter a number:
10
Factorial of 10 is: 3628800
[jayeshkumar@localhost ~]$ ./factorial.sh
Enter a number:
5
Factorial of 5 is: 120
[jayeshkumar@localhost ~]$
```

Factorial

18) How do you handle signals like Ctrl+C in a shell script?

In a shell script, you can handle signals like Ctrl+C (also known as SIGINT) using the trap command. Ctrl+C generates a SIGINT signal when the user presses it to interrupt the running script or program. By using the trap command, you can specify actions to be taken when a particular signal is received. Here's how you handle signals like Ctrl+C in a shell script:

```
#!/bin/bash
cleanup() {
    echo "Script interrupted. Performing cleanup..."
    # Add your cleanup actions here
    exit 1
}
# Set up a trap to call the cleanup function when Ctrl+C (SIGINT) is
received
trap cleanup SIGINT
# Rest of your script
echo "Running..."
sleep 10
echo "Finished."
```

Handling signals is important for making scripts robust and ensuring that they handle unexpected interruptions gracefully. You can customize the

cleanup function to match your specific needs, like closing files, stopping processes, or logging information before the script exits.

Explanation:

1. `#!/bin/bash`: This shebang line specifies the interpreter to be used for running the script.
2. `cleanup() { ... }`: This defines a function named `cleanup`. Inside this function, you can include any actions that need to be performed when the script is interrupted, such as closing files, releasing resources, or performing other cleanup tasks.
3. `trap cleanup SIGINT`: The `trap` command is used to set up a signal handler. In this case, it specifies that when the `SIGINT` signal (Ctrl+C) is received, the `cleanup` function should be executed.
4. `echo "Running..."`, `sleep 10`, `echo "Finished."`: These are just sample commands to simulate the execution of a script.

```
[jayeshkumar@localhost ~]$ vim handel.sh
[jayeshkumar@localhost ~]$ chmod +x handel.sh
[jayeshkumar@localhost ~]$ ./handel.sh
Running...
^CScript interrupted. Performing cleanup...
[jayeshkumar@localhost ~]$
```

19) Create a script that checks for and removes duplicate lines in a text file.

Here is our linux script in which we will remove duplicate lines from a text file.

```
#!/bin/bash
input_file="input.txt"
output_file="output.txt"
sort "$input_file" | uniq > "$output_file"
echo "Duplicate lines removed successfully."
```

Explanation:

1. The script starts with a shebang (`#!/bin/bash`), which indicates that the script should be interpreted using the Bash shell.
2. The `input_file` variable is set to the name of the input file containing duplicate lines (change this to your actual input file name).
3. The `output_file` variable is set to the name of the output file where the duplicates will be removed (change this to your desired output file name).
4. The script uses the `sort` command to sort the lines in the input file. Sorting the lines ensures that duplicate lines are grouped together.
5. The sorted lines are then passed through the `uniq` command, which removes consecutive duplicate lines. The output of this process is redirected to the output file.
6. After the duplicates are removed, the script prints a success message.

```
[jayeshkumar@localhost ~]$ cat input.txt
hello, hi , how , bye , hello
hello, hi , how , bye , hello
hello, 234 , how, bye , hello

[jayeshkumar@localhost ~]$ cat output.txt
[jayeshkumar@localhost ~]$ vim duplicate.sh
[jayeshkumar@localhost ~]$ chmod +x duplicate.sh
[jayeshkumar@localhost ~]$ ./duplicate.sh
Duplicate lines removed successfully.
[jayeshkumar@localhost ~]$ cat output.txt

hello, 234 , how, bye , hello
hello, hi , how , bye , hello
[jayeshkumar@localhost ~]$
```

duplicate line removing

Here, we use ``cat`` to display the text inside the text file.

20) Write a script that generates a secure random password.

Here is our script to generate a secure random password.

```
#!/bin/bash
# Function to generate a random password
generate_password() {
tr -dc 'A-Za-z0-9!@#$%^&*()_+{}[]' < /dev/urandom | fold -w 12 |
head -n 1
}
# Call the function and store the generated password
password=$(generate_password)
echo "Generated password: $password"
```

Note: User can accordingly change the length of there password, by replacing the number `12`.

Explanation:

1. The script starts with a shebang (`#!/bin/bash`), indicating that it should be interpreted using the Bash shell.
2. The `generate_password` function is defined to generate a random password. Here's how it works:
 - `tr -dc 'A-Za-z0-9!@#$%^&*()_+{}[]' < /dev/urandom` uses the `tr` command to delete (-d) characters not in the specified set of characters (`A-Za-z0-9!@#$%^&*()_+{}[]`) from the random data provided by `/dev/urandom`.
 - `fold -w 12` breaks the filtered random data into lines of width 12 characters each.
 - `head -n 1` selects the first line, effectively giving us a random sequence of characters of length 12.
3. The `password` variable is assigned the result of calling the `generate_password` function.
4. Finally, the generated password is displayed using `echo`.

```
[jayeshkumar@localhost ~]$ vim Generate_password.sh
[jayeshkumar@localhost ~]$ chmod +x Generate_password.sh
[jayeshkumar@localhost ~]$ ./Generate_password.sh
Generated password: q64jg!Rc&Bc7
[jayeshkumar@localhost ~]$ ./Generate_password.sh
Generated password: 2edkF(SGMA{G
[jayeshkumar@localhost ~]$ ./Generate_password.sh
Generated password: jLJDTurz]Mt[
[jayeshkumar@localhost ~]$
```

21) Write a shell script that calculates the total size of all files in a directory.

Here is a shell script to calculate the total size of all files in a directory.

```
#!/bin/bash
directory="/path/to/your/directory"
total_size=$(du -csh "$directory" | grep total | awk '{print $1}')
echo "Total size of files in $directory: $total_size"
```

Explanation:

1. The script starts with the `#!/bin/bash` shebang, indicating that it should be interpreted using the Bash shell.
2. The `directory` variable is set to the path of the directory for which you want to calculate the total file size. Replace `"/path/to/your/directory"` with the actual path.
3. The `du` command is used to estimate file space usage. The options used are:
 - `-c`: Produce a grand total.
 - `-s`: Display only the total size of the specified directory.
 - `-h`: Print sizes in a human-readable format (e.g., KB, MB, GB).
4. The output of `du` is piped to `grep total` to filter out the line that contains the total size.
5. `awk '{print $1}'` is used to extract the first field (total size) from the line.

6. The calculated total size is stored in the total_size variable.
7. Finally, the script displays the total size using echo.

```
[jayeshkumar@localhost ~]$ pwd
/home/jayeshkumar
[jayeshkumar@localhost ~]$ vim size.sh
[jayeshkumar@localhost ~]$ chmod +x size.sh
[jayeshkumar@localhost ~]$ ./size.sh
Total size of files in /home/jayeshkumar: 123M
[jayeshkumar@localhost ~]$
```

Total Size of Files

Here , we used ``pwd`` command to see the current directory path.

22) Explain the difference between if and elif statements in shell scripting.

Feature	<code>`if`</code> Statement	<code>`elif`</code> Statement
Purpose	Explain the difference between if and elif statements in shell scripting.	Provides alternative conditions to check when the initial if condition is false.
usage	Used for the initial condition.	Used after the initial if condition to check additional conditions.
number of Blocks	Can have only one if block.	Can have multiple elif blocks, but only one else block (optional).
Execution	Executes the block of code associated with the if statement if the condition is true. If the condition is false,	Checks each elif condition in order. If one elif condition is true, the corresponding block of code is executed, and the script exits the entire conditional

Feature	`if` Statement	`elif` Statement
	the else block (if present) is executed (optional).	block. If none of the elif conditions are true, the else block (if present) is executed.
Nested Structures	Can be nested within other if, elif, or else blocks.	Cannot be nested within another elif block, but can be used inside an if or else block.

Lets understand it by an example.

```
#!/bin/bash
number=5
if [ $number -gt 10 ]; then
echo "$number is greater than 10"
else
echo "$number is not greater than 10"
fi
echo "-----"
if [ $number -gt 10 ]; then
echo "$number is greater than 10"
elif [ $number -eq 10 ]; then
echo "$number is equal to 10"
else
echo "$number is less than 10"
fi
```

Explanation:

In this example, the first if block checks whether number is greater than 10. If not, it prints a message indicating that the number is not greater than 10. The second block with elif statements checks multiple conditions sequentially until one of them is true. In this case, since the value of number is 5, the output will be:


```
[jayeshkumar@localhost ~]$ vim if_elif.sh
[jayeshkumar@localhost ~]$ chmod +x if_elif.sh
[jayeshkumar@localhost ~]$ ./if_elif.sh
5 is not greater than 10
-----
5 is less than 10
[jayeshkumar@localhost ~]$
```

if_elif difference

23) How do you use a while loop to repeatedly execute commands?

A while loop is used in shell scripting to repeatedly execute a set of commands as long as a specified condition is true. The loop continues executing the commands until the condition becomes false.

Here's the basic syntax of a while loop:

```
while [ condition ]; do
# Commands to be executed
done
```

Explanation:

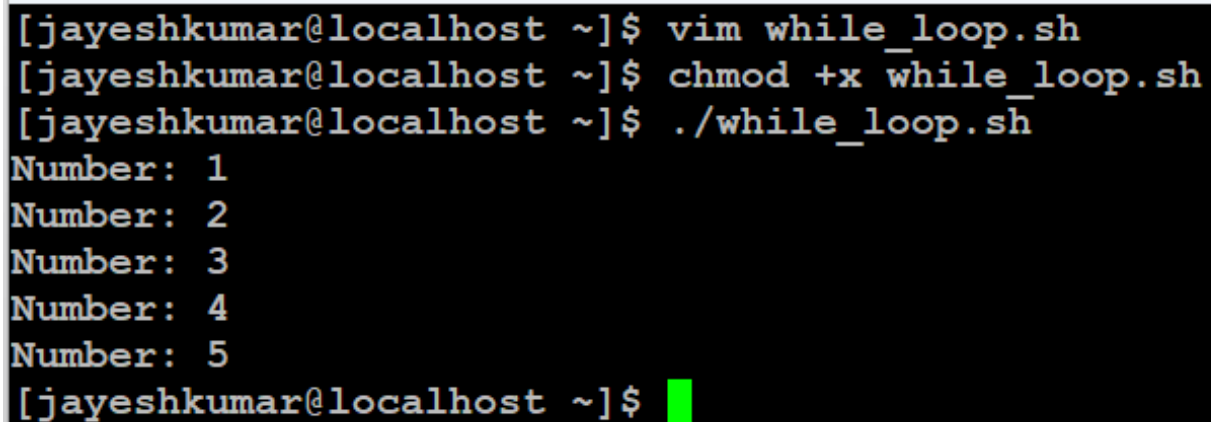
1. The `while` loop starts with the keyword `while` followed by a condition enclosed within square brackets `[]`.
2. The loop's body, which contains the commands to be executed, is enclosed within the `do` and `done` keywords.
3. The loop first checks the condition. If the condition is true, the commands within the loop body are executed. After the loop body executes, the condition is checked again, and the process repeats until the condition becomes false.

Example: If we want to print numbers from 1 to 5


```
#!/bin/bash
counter=1
while [ $counter -le 5 ]; do
echo "Number: $counter"
counter=$((counter + 1))
done
```

Explanation:

- The counter variable is set to 1.
- The while loop checks whether the value of counter is less than or equal to 5. As long as this condition is true, the loop continues executing.
- Inside the loop, the current value of counter is printed using echo.
- The counter is incremented by 1 using the expression `$((counter + 1))`.

A terminal window with a black background and yellow text. The prompt is [jayeshkumar@localhost ~]\$. The user enters vim while_loop.sh, then chmod +x while_loop.sh, and finally ./while_loop.sh. The output shows five lines: Number: 1, Number: 2, Number: 3, Number: 4, and Number: 5. The prompt returns to [jayeshkumar@localhost ~]\$.

```
[jayeshkumar@localhost ~]$ vim while_loop.sh
[jayeshkumar@localhost ~]$ chmod +x while_loop.sh
[jayeshkumar@localhost ~]$ ./while_loop.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
[jayeshkumar@localhost ~]$
```

while loop

24) Create a shell script that finds and lists all empty files in a directory.

Shell script that you can use to find and list all empty files in a directory using the `find` and `stat` commands:

```
#!/bin/bash
directory="$1"
if [ -z "$directory" ]; then
echo "Usage: $0 <directory>"
```

```
exit 1
fi

if [ ! -d "$directory" ]; then
echo "Error: '$directory' is not a valid directory."
exit 1
fi
echo "Empty files in $directory:"
find "$directory" -type f -empty
```

Explanation:

1. `#!/bin/bash`: This is called a "shebang," and it tells the operating system to use the Bash shell to interpret and execute the script.
2. `directory="$1"`: This line assigns the first command-line argument (denoted by `$1`) to the variable `directory`.
3. `if [-z "$directory"]; then`: This line starts an if statement that checks whether the `directory` variable is empty (`-z` tests for an empty string).
4. `echo "Usage: $0 <directory>"`: If the directory is empty, this line prints a usage message, where `$0` represents the script's name.
5. `exit 1`: This line exits the script with an exit code of `1`, indicating an error.
6. `fi`: This line marks the end of the `if` statement.
7. `if [! -d "$directory"]; then`: This starts another if statement to check if the provided directory exists (`-d` tests for a directory).
8. `echo "Error: '$directory' is not a valid directory."`: If the provided directory doesn't exist, this line prints an error message.
9. `exit 1`: Exits the script with an exit code of `1`.
10. `fi`: Marks the end of the second `if` statement.
11. `echo "Empty files in $directory:"`: If everything is valid so far, this line prints a message indicating that the script will list empty files in the specified directory.

12. `find "$directory" -type f -empty`: This line uses the `find` command to search for empty files (`-empty`) of type regular files (`-type f`) in the specified directory. It then lists these empty files.

```
[jayeshkumar@localhost ~]$ vim empty_files.sh
[jayeshkumar@localhost ~]$ chmod +x empty_files.sh
[jayeshkumar@localhost ~]$ ./empty_files.sh /home/jayeshkumar/
Empty files in /home/jayeshkumar/:
0 /home/jayeshkumar/.mozilla/firefox/cjpe0972.default-default/
0 /home/jayeshkumar/.config/.gsd-keyboard.settings-ported
0 /home/jayeshkumar/.config/enchant/en_US.dic
```

Finding empty files

Note: We need to provide a directory as an argument when running the script. Here we have used the path of current directory
 "home/jayeshkumar/"

25) What is the purpose of the read command in shell scripting?

The read command in shell scripting lets the script ask you for information. It's like when a computer asks you a question and waits for your answer. This is useful for scripts that need you to type something or for when the script needs to work with information from files. The read command helps the script stop and wait for what you type, and then it can use that information to do more things in the script.

Syntax of read command:

```
read variable_name
```

Example: If we want to take name as an input from user to print it.

```
#!/bin/bash
echo "Please enter your name:"
read name
echo "Hello, $name!"
```

```
[jayeshkumar@localhost ~]$ vim read_name.sh
[jayeshkumar@localhost ~]$ chmod +x read_name.sh
[jayeshkumar@localhost ~]$ ./read_name.sh
Please enter your name:
jayesh
Hello, jayesh!
[jayeshkumar@localhost ~]$
```

read nam

In summary, the read command is used to capture user input or data from files within shell scripts, making the scripts more interactive and versatile.

26) Write a shell script that converts all filenames in a directory to lowercase.

Here's a shell script that converts all filenames in a directory to lowercase.

```
#!/bin/bash
directory="$1"
if [ -z "$directory" ]; then
    echo "Usage: $0 <directory>"
    exit 1
fi

if [ ! -d "$directory" ]; then
    echo "Error: '$directory' is not a valid directory."
    exit 1
fi

cd "$directory" || exit 1

for file in *; do
    if [ -f "$file" ]; then
        newname=$(echo "$file" | tr 'A-Z' 'a-z')
        [ "$file" != "$newname" ] && mv "$file" "$newname"
    fi
done
```

fi
done

Explanation:

1. **#!/bin/bash**: This is the shebang, specifying that the script should be interpreted using the Bash shell.
2. **directory="\$1"**: This line assigns the first command-line argument to the variable `directory`.
3. **if [-z "\$directory"]; then**: This line checks if the `directory` variable is empty (no argument provided when running the script).
4. **echo "Usage: \$0 <directory>"**: If the `directory` is empty, this line prints a usage message with the script's name (`$0`).
5. **exit 1**: This line exits the script with an exit code of 1, indicating an error occurred.
6. **fi**: This marks the end of the first if statement.
7. **if [! -d "\$directory"]; then**: This line checks if the specified directory does not exist (`-d` tests for a directory).
8. **echo "Error: '\$directory' is not a valid directory."**: If the specified directory doesn't exist, this line prints an error message.
9. **exit 1**: Exits the script with an exit code of 1.
10. **fi**: Marks the end of the second if statement.
11. **cd "\$directory" || exit 1**: Changes the current working directory to the specified directory. If the directory change fails (e.g., non-existent directory), the script exits with an error code.
12. **for file in *; do** | **for file in *; do**: Initiates a loop that iterates over all items in the current directory (`*` matches all filenames).
13. **if [-f "\$file"]; then**: Checks if the current loop iteration item is a regular file (`-f` tests for a regular file).
14. **newname=\$(echo "\$file" | tr 'A-Z' 'a-z')**: Converts the current filename (`$file`) to lowercase using the `tr` command and stores the result in the `newname` variable.

15. ["\$file" != "\$newname"] && mv "\$file" "\$newname": Compares the original filename with the new lowercase filename. If they are different, it renames the file using the mv command.
16. **fi**: Marks the end of the inner if statement.
17. **done**: Marks the end of the loop.

```
[jayeshkumar@localhost test]$ ls
FILE_1  FILE_2  File_3
[jayeshkumar@localhost test]$ vim lower.sh
[jayeshkumar@localhost test]$ chmod +x lower.sh
[jayeshkumar@localhost test]$ ./lower.sh /home/jayeshkumar/test
[jayeshkumar@localhost test]$ ls
file_1  file_2  file_3  lower.sh
[jayeshkumar@localhost test]$
```

Note: We need to provide a directory as an argument when running the script. Here we have used the path of current directory "home/jayeshkumar/test"

27) How can you use arithmetic operations within a shell script?

Arithmetic operations can be performed within a shell script using various built-in methods. The shell provides mechanisms for simple arithmetic calculations using arithmetic expansion Like:

1. Arithmetic Expansion ($\$(())$)
2. Using **expr** Command
3. Using **let** Command

Here is our Shell script explaining all three methods for arithmetic operations.

```
#!/bin/bash
num1=10
num2=5
```

#Arithmetic Expansion ($\$(())$)

```
result=$((num1 + num2))  
echo "Sum: $result"
```

#Using expr Command

```
sum=$(expr $num1 + $num2)  
echo "Sum: $sum"
```

#Using let Command

```
let "sum = num1 + num2"  
echo "Sum: $sum"
```

Explanation:

1. **`#!/bin/bash`**: This is the shebang, specifying that the script should be interpreted using the Bash shell.
2. **`num1=10`** and **`num2=5`**: These lines assign the values 10 and 5 to the variables **`num1`** and **`num2`**, respectively.
3. **`#Arithmetic Expansion (\$((...)))`**: This is a comment indicating the start of the section that demonstrates arithmetic expansion.
4. **`result=\$((num1 + num2))`**: This line uses arithmetic expansion to calculate the sum of **`num1`** and **`num2`** and stores the result in the **`result`** variable.
5. **`echo "Sum: \$result"`**: This line prints the calculated sum using the value stored in the **`result`** variable.
6. **`#Using expr Command`**: This is a comment indicating the start of the section that demonstrates using the **`expr`** command for arithmetic operations.

7. ``sum=$(expr $num1 + $num2)``: This line uses the ``expr`` command to calculate the sum of ``num1`` and ``num2`` and stores the result in the ``sum`` variable. Note that the ``expr`` command requires spaces around the operators.
8. ``echo "Sum: $sum"``: This line prints the calculated sum using the value stored in the ``sum`` variable.
9. ``#Using let Command``: This is a comment indicating the start of the section that demonstrates using the ``let`` command for arithmetic operations.
10. ``let "sum = num1 + num2"``: This line uses the ``let`` command to calculate the sum of ``num1`` and ``num2`` and assigns the result to the ``sum`` variable. The ``let`` command does not require spaces around the operators.
11. ``echo "Sum: $sum"``: This line prints the calculated sum using the value stored in the ``sum`` variable.

```
[jayeshkumar@localhost test]$ vim arithmetic.sh
[jayeshkumar@localhost test]$ chmod +x arithmetic.sh
[jayeshkumar@localhost test]$ ./arithmetic.sh
Sum: 15
Sum: 15
Sum: 15
[jayeshkumar@localhost test]$
```

arithmetic

28) Create a script that checks if a network host is reachable.

Here's a simple shell script that uses the ping command to check if a network host is reachable:

```
#!/bin/bash
host="$1"
if [ -z "$host" ]; then
echo "Usage: $0 <hostname or IP>"
exit 1
fi
ping -c 4 "$host"
```



```
if [ $? -eq 0 ]; then
echo "$host is reachable."
else
echo "$host is not reachable."
fi
```

Explanation:

1. It takes a hostname or IP address as an argument and checks if the argument is provided.
2. If no argument is provided, it displays a usage message and exits.
3. It uses the ping command with the -c 4 option to send four ICMP echo requests to the specified host.
4. After the ping command runs, it checks the exit status (\$?). If the exit status is 0, it means the host is reachable and the script prints a success message. Otherwise, it prints a failure message.

```
[jayeshkumar@localhost test]$ vim network.sh
[jayeshkumar@localhost test]$ chmod +x network.sh
[jayeshkumar@localhost test]$ ./network.sh google.com
PING google.com (142.250.77.206) 56(84) bytes of data.
64 bytes from del11s08-in-f14.1e100.net (142.250.77.206): icmp_seq=1 ttl=116 time=6.57 ms
64 bytes from del11s08-in-f14.1e100.net (142.250.77.206): icmp_seq=2 ttl=116 time=5.20 ms
64 bytes from del11s08-in-f14.1e100.net (142.250.77.206): icmp_seq=3 ttl=116 time=1.32 ms
64 bytes from del11s08-in-f14.1e100.net (142.250.77.206): icmp_seq=4 ttl=116 time=1.36 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 4546ms
rtt min/avg/max/mdev = 1.320/3.611/6.569/2.322 ms
google.com is reachable.
[jayeshkumar@localhost test]$
```

Note: We need to provide a hostname as an argument when running the script. Here we have used "google.com"

29) Write a Shell Script to Find the Greatest Element in an Array:

Here's a shell script to find the greatest element in an array.

```
#!/bin/bash
# Declare an array
```

```
array=(3 56 24 89 67)
```

```
# Initialize a variable to store the maximum value, starting with the  
first element
```

```
max=${array[0]}
```

```
# Iterate through the array
```

```
for num in "${array[@]}; do
```

```
# Compare each element with the current maximum
```

```
    if ((num > max)); then  
        max=$num  
    fi  
done
```

```
# Print the maximum value
```

```
echo "The maximum element in the array is: $max"
```

Explanation:

1. ``#!/bin/bash``: The shebang line specifies that the script should be interpreted using the Bash shell.
2. ``array=(3 56 24 89 67)``: The array is declared and initialized with values.

3. ``max=${array[0]}``: ``max`` is initialized with the first element of the array.
4. ``for num in "${array[@]}"; do``: A ``for`` loop is used to iterate through the elements of the array.
5. ``if ((num > max)); then``: An ``if`` statement checks if the current element ``num`` is greater than the current maximum ``max``.
6. ``max=$num``: If ``num`` is greater than ``max``, ``max`` is updated with the value of `num`.
7. ``done``: The ``for`` loop is closed.
8. ``echo "The maximum element in the array is: $max"``: Finally, the script prints the maximum value found in the array.

```
[jayeshkumar@localhost ~]$ vim max_array.sh
[jayeshkumar@localhost ~]$ chmod +x max_array.sh
[jayeshkumar@localhost ~]$ ./max_array.sh
The maximum element in the array is: 89
[jayeshkumar@localhost ~]$
```

greatest number

30) Write a script to calculate the sum of Elements in an Array.

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

```
# Declare an array
```

```
array=(1 65 22 19 94)
```

```
# Initialize a variable to store the sum
```

```
sum=0
```

Iterate through the array and add each element to the sum

```
for num in "${array[@]}; do
    sum=$((sum + num))
done
```

Print the sum

echo "The sum of elements in the array is: \$sum"

Explanation:

`#!/bin/bash`: The shebang line specifies that the script should be interpreted using the Bash shell.

`array=(1 65 22 19 94)`: The array is declared and initialized with values.

`sum=0`: **`sum`** is initialized to zero to hold the sum of elements.

`for num in "\${array[@]}; do`: A **`for`** loop is used to iterate through the elements of the array.

`sum=\$((sum + num))`: Inside the loop, each element **`num`** is added to the **`sum`** variable.

`done`: The **`for`** loop is closed.

`echo "The sum of elements in the array is: \$sum"`: Finally, the script prints the sum of all elements in the array.

```
[jayeshkumar@localhost ~]$ vim sum_array.sh
[jayeshkumar@localhost ~]$ chmod +x sum_array.sh
[jayeshkumar@localhost ~]$ ./sum_array.sh
The sum of elements in the array is: 201
[jayeshkumar@localhost ~]$
```

Sum of Elements

Know More About Shell Scripts

- [Difference between shell and kernel](#)
- [Difference Between Bind Shell and Reverse Shell](#)
- [Introduction to Linux Shell and Shell Scripting](#)

Conclusion

We all geeks know that shell script is very useful to increases the work productivity as well as it save time also. So, in this article we have covered **30 very useful and most conman shell scripts examples**. We hope that this complete guide on shell scripting example help you to understand the all about the shell scripts.

Master DevOps and also get 90% Course fee refund on completing 90% course in 90 days! [Take the Three 90 Challenge today.](#)

The most sought-after Three 90 challenge has started and this is your chance to upskill and get 90% refund. What more motivation do you need? [Start the challenge right away!](#)

Comment

More info

Advertise with us

Next Article

Menu-Driven Shell Script

Similar Reads

Shell Script to Show the Difference Between echo “\$SHELL” and echo...

In shell scripting and Linux, the echo command is used to display text on the terminal or console. When used with the \$SHELL variable, which contains...

4 min read

Bash Script - Difference between Bash Script and Shell Script

In computer programming, a script is defined as a sequence of instructions that is executed by another program. A shell is a command-line interpreter ...

3 min read

Shell Scripting - Difference between Korn Shell and Bash shell

Korn Shell: Korn Shell or KSH was developed by a person named David Korn, which attempts to integrate the features of other shells like C shell, Bourne...

3 min read

Shell Script to Demonstrate the Use of Shell Function Library

Shell Function Library is basically a collection of functions that can be accessed from anywhere in the development environment. It actually make...

3 min read

How to write a shell script that starts tmux session, and then runs a ru...

Scripts are important for streamlining workflows and automating tasks. One such way to streamline workflow is by using Tmux [Terminal Multiplexer]....

4 min read

Introduction to Linux Shell and Shell Scripting

If we are using any major operating system, we are indirectly interacting with the shell. While running Ubuntu, Linux Mint, or any other Linux distribution,...

8 min read

Shell Scripting - Default Shell Variable Value

A shell gives us an interface to the Unix system. While using an operating system, we indirectly interact with the shell. On Linux distribution systems,...

3 min read

Shell Scripting - Interactive and Non-Interactive Shell

A shell gives us an interface to the Unix system. While using an operating system, we indirectly interact with the shell. On Linux distribution systems,...

3 min read

Korn Shell vs Bash Shell

Introduction : Korn Shell, also known as ksh, is a Unix shell that was developed by David Korn in the early 1980s. It was designed to be a more...

6 min read

How to protect Linux shell file using noclobber in bash shell?

Most Linux shells(bash, csh, ksh, tcsh) have a built-in file protection mechanism to prevent files from being overwritten accidentally. In this...

2 min read



Corporate & Communications Address:

A-143, 7th Floor, Sovereign Corporate
Tower, Sector- 136, Noida, Uttar Pradesh
(201305)

Registered Address:

K 061, Tower K, Gulshan Vivante
Apartment, Sector 137, Noida, Gautam
Buddh Nagar, Uttar Pradesh, 201305



Advertise with us

Company

About Us
Legal
Privacy Policy
Careers
In Media
Contact Us
GFG Corporate Solution
Placement Training Program

Languages

Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial

Data Science & ML

Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy
NLP
Deep Learning

Python Tutorial

Python Programming Examples
Django Tutorial
Python Projects
Python Tkinter

Explore

Job-A-Thon Hiring Challenge
Hack-A-Thon
GfG Weekly Contest
Offline Classes (Delhi/NCR)
DSA in JAVA/C++
Master System Design
Master CP
GeeksforGeeks Videos
Geeks Community

DSA

Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
DSA Interview Questions
Competitive Programming

Web Technologies

HTML
CSS
JavaScript
TypeScript
ReactJS
NextJS
NodeJs
Bootstrap
Tailwind CSS

Computer Science

GATE CS Notes
Operating Systems
Computer Network
Database Management System

Web Scraping
OpenCV Tutorial
Python Interview Question

Software Engineering
Digital Logic Design
Engineering Maths

DevOps

Git
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

System Design

High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

School Subjects

Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar

Commerce

Accountancy
Business Studies
Economics
Management
HR Management
Finance
Income Tax

Databases

SQL
MYSQL
PostgreSQL
PL/SQL
MongoDB

Preparation Corner

Company-Wise Recruitment Process
Resume Templates
Aptitude Preparation
Puzzles
Company-Wise Preparation
Companies
Colleges

Competitive Exams

JEE Advanced
UGC NET
UPSC
SSC CGL
SBI PO
SBI Clerk
IBPS PO
IBPS Clerk

More Tutorials

Software Development
Software Testing
Product Management
Project Management
Linux
Excel
All Cheat Sheets
Recent Articles

Free Online Tools

Typing Test
Image Editor
Code Formatters

Write & Earn

Write an Article
Improve an Article
Pick Topics to Write

[Code Converters](#)[Currency Converter](#)[Random Number Generator](#)[Random Password Generator](#)[Share your Experiences](#)[Internships](#)

DSA/Placements

[DSA - Self Paced Course](#)[DSA in JavaScript - Self Paced Course](#)[DSA in Python - Self Paced](#)[C Programming Course Online - Learn C with Data Structures](#)[Complete Interview Preparation](#)[Master Competitive Programming](#)[Core CS Subject for Interview Preparation](#)[Mastering System Design: LLD to HLD](#)[Tech Interview 101 - From DSA to System Design \[LIVE\]](#)[DSA to Development \[HYBRID\]](#)[Placement Preparation Crash Course \[LIVE\]](#)

Development/Testing

[JavaScript Full Course](#)[React JS Course](#)[React Native Course](#)[Django Web Development Course](#)[Complete Bootstrap Course](#)[Full Stack Development - \[LIVE\]](#)[JAVA Backend Development - \[LIVE\]](#)[Complete Software Testing Course \[LIVE\]](#)[Android Mastery with Kotlin \[LIVE\]](#)

Machine Learning/Data Science

[Complete Machine Learning & Data Science Program - \[LIVE\]](#)[Data Analytics Training using Excel, SQL, Python & PowerBI - \[LIVE\]](#)[Data Science Training Program - \[LIVE\]](#)[Mastering Generative AI and ChatGPT](#)[Data Science Course with IBM Certification](#)

Programming Languages

[C Programming with Data Structures](#)[C++ Programming Course](#)[Java Programming Course](#)[Python Full Course](#)

Clouds/Devops

[DevOps Engineering](#)[AWS Solutions Architect Certification](#)[Salesforce Certified Administrator Course](#)

GATE 2026

[GATE CS Rank Booster](#)[GATE DA Rank Booster](#)[GATE CS & IT Course - 2026](#)[GATE DA Course 2026](#)[GATE Rank Predictor](#)

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved