Bash Scripting

Waheed Iqbal

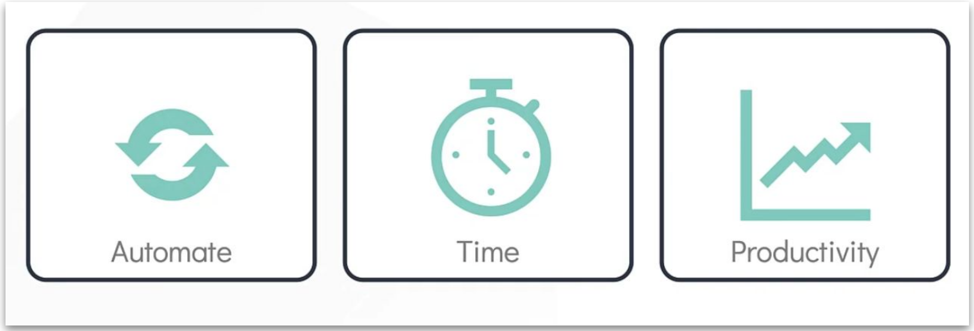
DevOps (Fall 2023)
Department of Data Science, FCIT, University of the Punjab
Lahore, Pakistan

Introduction

- Bash (**B**ourne **A**gain **SH**ell) is a Unix shell and command-line interpreter.
- It is used to execute commands and scripts.
- Bash is useful for automating tasks, processing and analyzing data, and managing system resources.
- It is widely available on many different operating systems.
- You may often hear shell scripting or bash scripting interchangeably.
- Bash is a specific type of shell, while shell is a general term.

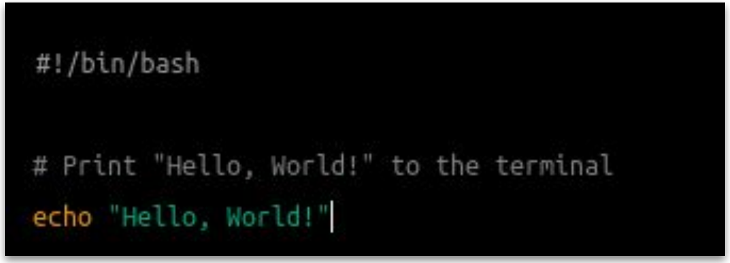
Why Bash Scripting

- To automate repetitive tasks, which can save time and improve efficiency.
- To perform a wide range of tasks, including interacting with the operating system, monitoring system resources, and processing data.
- Bash scripts are portable, which means they can be run on any Unix-like operating system, including Linux and macOS.
- Easy-to-learn syntax, which makes it a good choice for beginners who are new to programming.

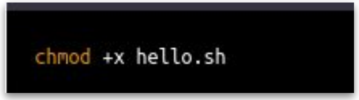


Writing Simple Bash scripts

- Create a new script file named hello.sh
- Make the script executable
- Then, you can run the script by typing ./hello.sh at the command line. This will print "Hello, World!" to the terminal.
- Lets try it!



```
#!/bin/bash

# Print "Hello, World!" to the terminal
echo "Hello, World!"
```

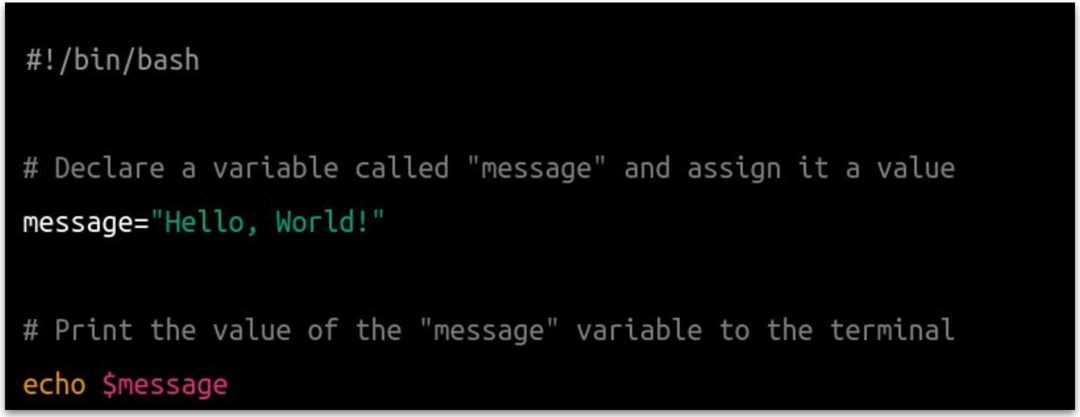


```
chmod +x hello.sh
```

hello.sh include the **shebang** line (#!/bin/bash) and a command (echo) to print the string on the terminal

Variables in Bash Script

- Bash scripts use variables to store and manipulate data.
- Bash scripts can accept input from the command line in the form of arguments.
- Lets create a simple script with one variable and print the value of the script on the terminal:



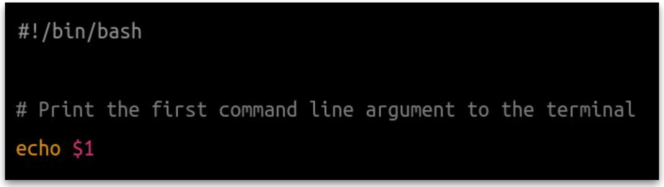
```bash
#!/bin/bash

# Declare a variable called "message" and assign it a value
message="Hello, World!"

# Print the value of the "message" variable to the terminal
echo $message
```

Variable Conventions

- Variable names must begin with a letter or an underscore, and can only contain letters, numbers, and underscores.
- For example, my_variable and VAR1 are valid variable names, but 1var is not.
- Bash is case-sensitive, so variables named VAR1, Var1, and var1 are considered to be different variables.
- Bash does not support data types, so all variables are treated as strings.
- Some variable names are reserved for special purposes in Bash. For example, $0 refers to the name of the script itself, and $1, $2, etc. refer to the command line arguments passed to the script.
- In Bash, the $(( )) syntax is used to perform arithmetic operations on variables.

Command Line Arguments

- Command line arguments are input that is passed to a command when it is run from the command line.
- They are often used to control the behavior of the command or script, or to provide it with data that it needs to operate.
- In Bash scripts, command line arguments are accessed using the $1, $2, $3, etc. variables. For example, consider the following script (script.sh):



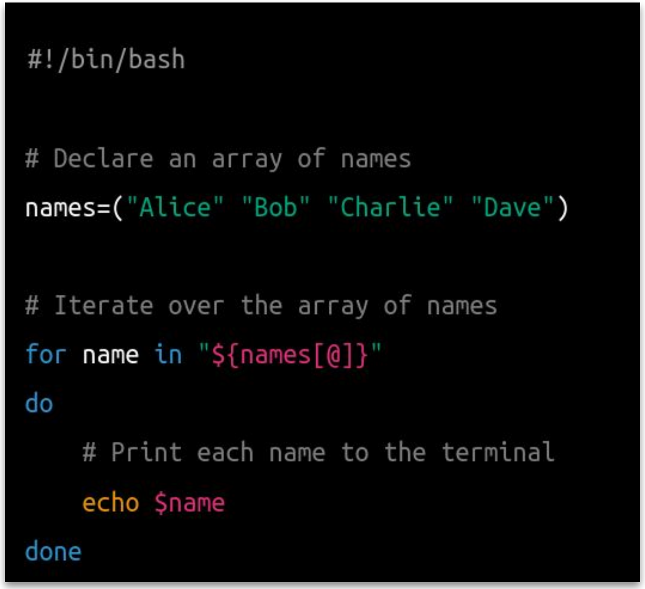
```
#!/bin/bash

# Print the first command line argument to the terminal
echo $1
```

```
./script.sh Hello
```

- Change the script to
  echo $@
- Run the script now again
  ./script.sh Hello World From Pakistan

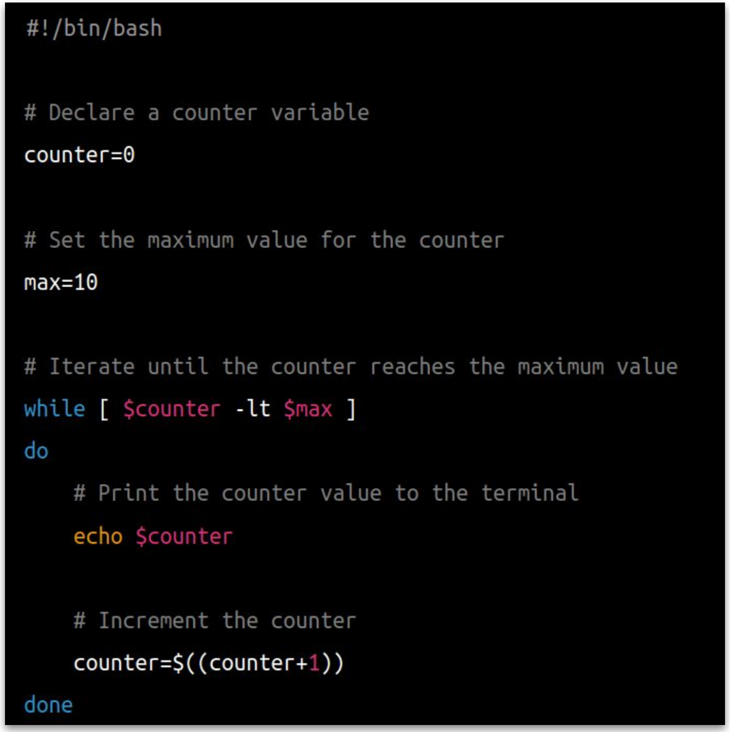
Conditional Statements and Looping

- In Bash, you can use for and while loops



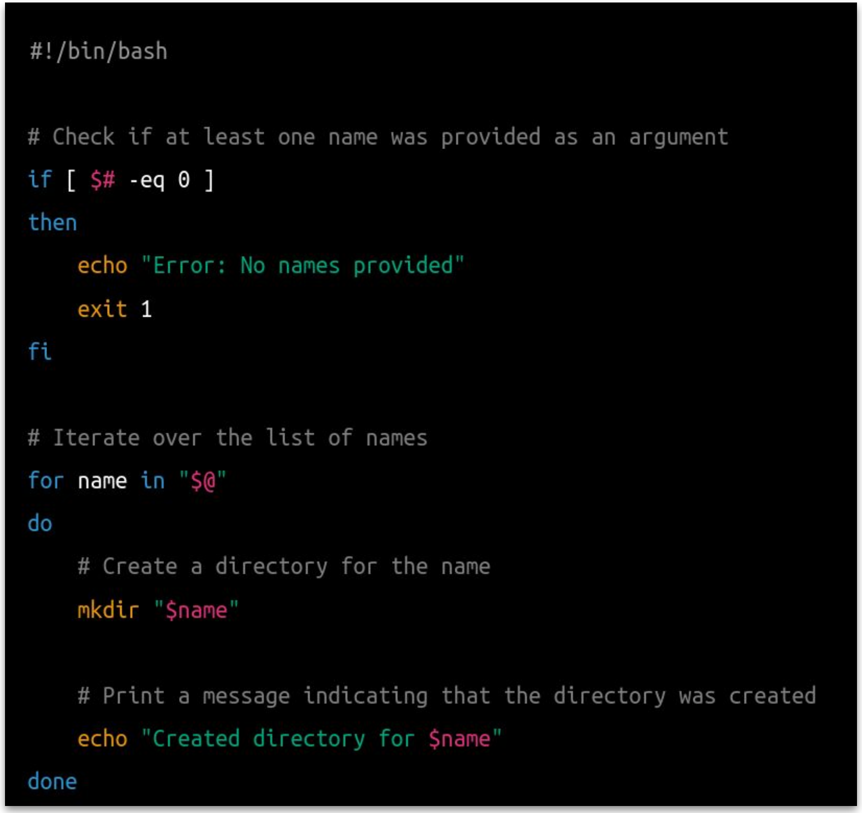
```
#!/bin/bash

# Declare an array of names
names=("Alice" "Bob" "Charlie" "Dave")

# Iterate over the array of names
for name in "${names[@]}"
do
    # Print each name to the terminal
    echo $name
done
```

The ${names[@]} syntax expands the names array into a list of its individual elements.



```
#!/bin/bash

# Declare a counter variable
counter=0

# Set the maximum value for the counter
max=10

# Iterate until the counter reaches the maximum value
while [ $counter -lt $max ]
do
    # Print the counter value to the terminal
    echo $counter

    # Increment the counter
    counter=$((counter+1))
done
```

Conditional Statements and Looping

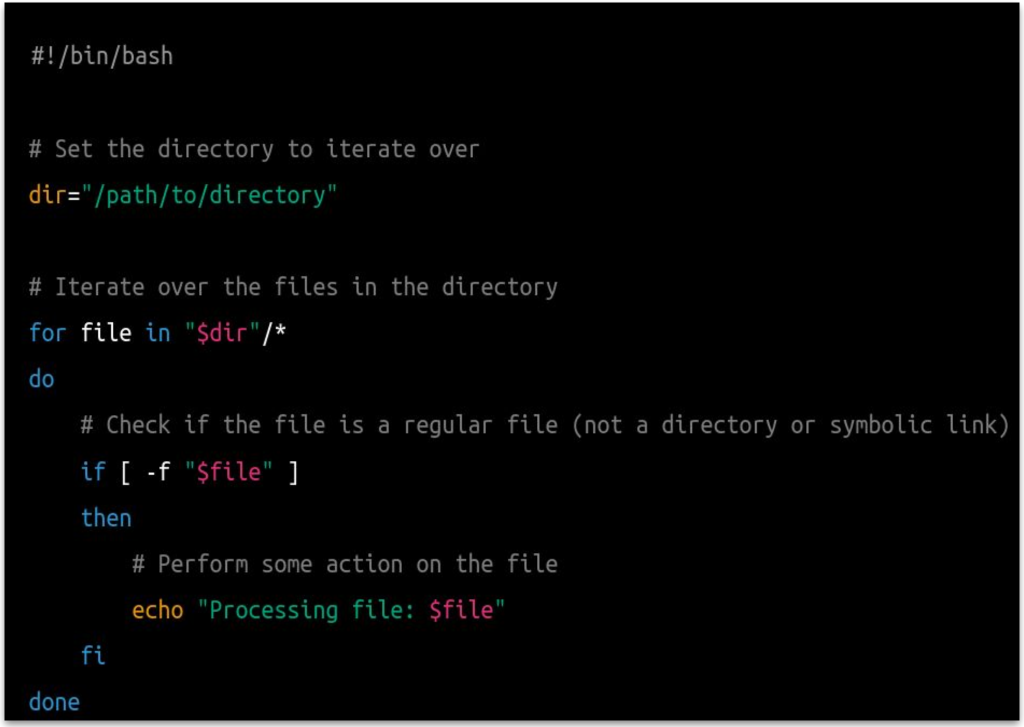


```
#!/bin/bash

# Check if at least one name was provided as an argument
if [ $# -eq 0 ]
then
    echo "Error: No names provided"
    exit 1
fi

# Iterate over the list of names
for name in "$@"
do
    # Create a directory for the name
    mkdir "$name"

    # Print a message indicating that the directory was created
    echo "Created directory for $name"
done
```

- It create the directories in the current location using the provided list through command line. For example:
    - ./script.sh dir1 dir2 dir3

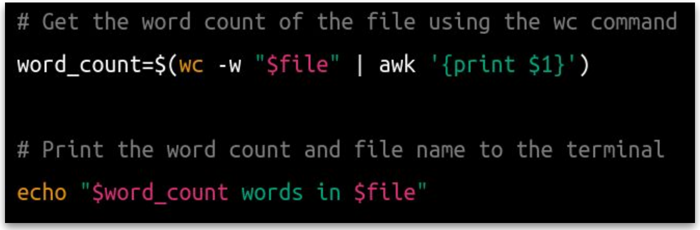
  will create three directorie in the current location.
- $# -eq 0 to check if the number of arguments are equal to zero. We can use other comparison too:
    - -eq: Equal to
    - -ne: Not equal to
    - -lt: Less than
    - -le: Less than or equal to
    - -gt: Greater than
    - -ge: Greater than or equal to

Conditional Statements and Looping (Cont.)



```
#!/bin/bash

# Set the directory to iterate over
dir="/path/to/directory"

# Iterate over the files in the directory
for file in "$dir"/*
do
    # Check if the file is a regular file (not a directory or symbolic link)
    if [ -f "$file" ]
    then
        # Perform some action on the file
        echo "Processing file: $file"
    fi
done
```

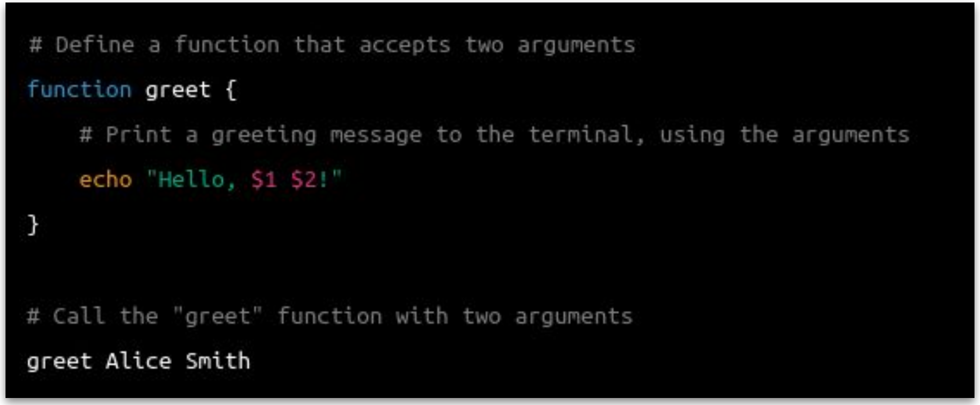
Lets update the script to count the words in each file:



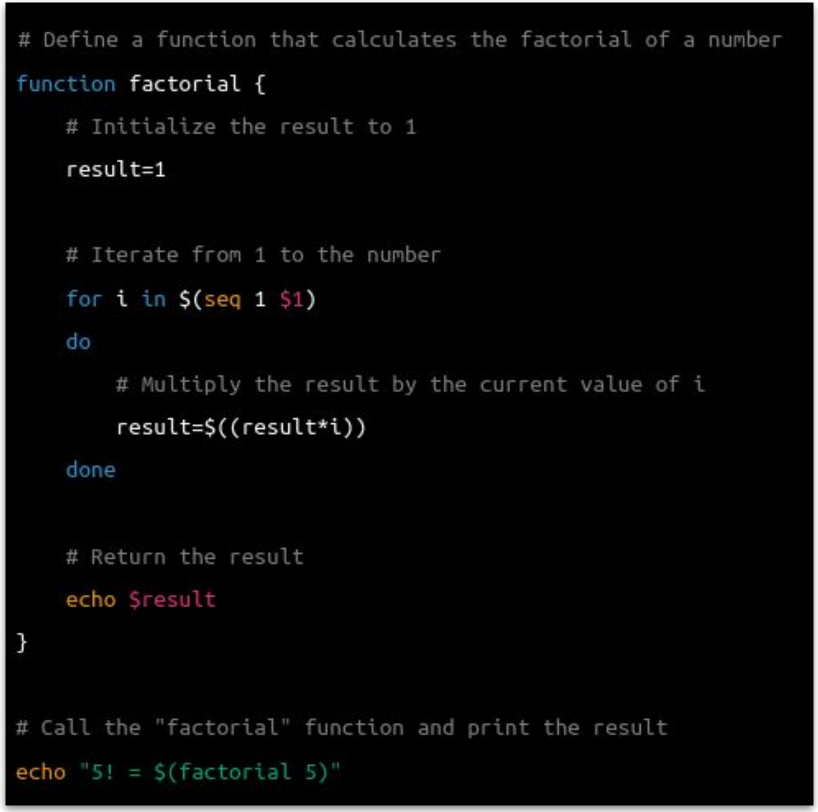
```
# Get the word count of the file using the wc command
word_count=$(wc -w "$file" | awk '{print $1}')

# Print the word count and file name to the terminal
echo "$word_count words in $file"
```

awk is a powerful command-line utility used to perform text processing on text files. It is particularly useful for processing tabular data, such as CSV or TSV files.

Functions

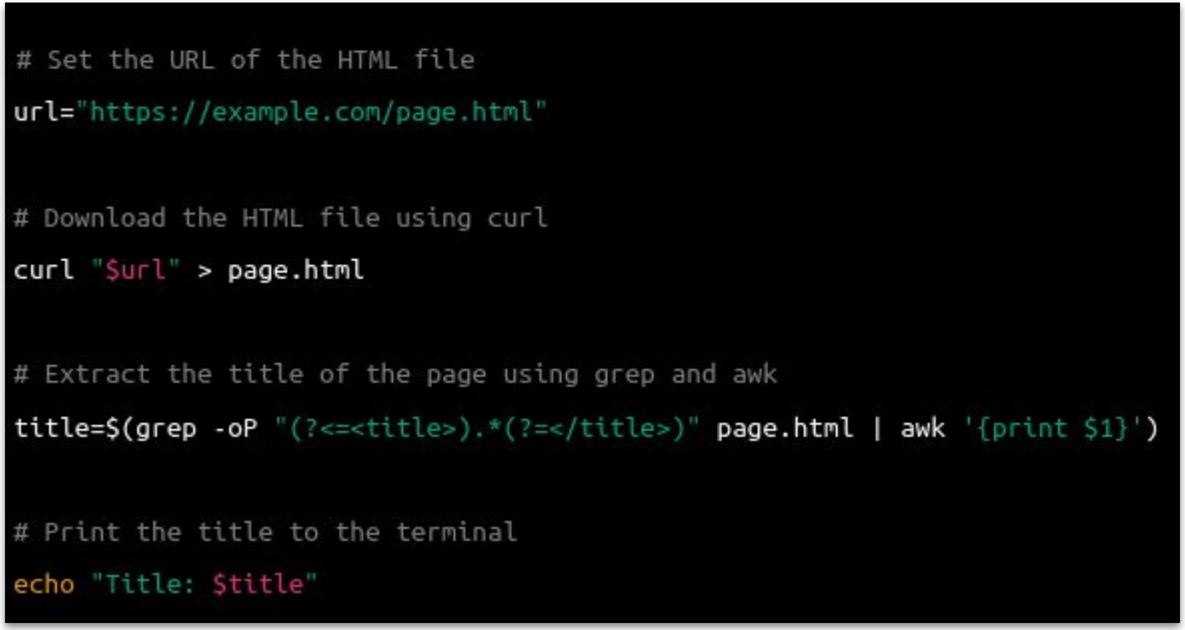


```bash
# Define a function that accepts two arguments
function greet {
    # Print a greeting message to the terminal, using the arguments
    echo "Hello, $1 $2!"
}

# Call the "greet" function with two arguments
greet Alice Smith
```



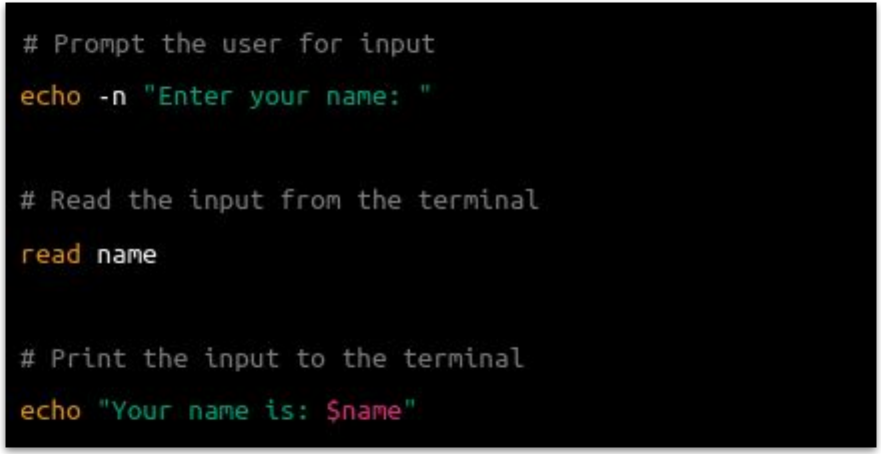
```bash
# Define a function that calculates the factorial of a number
function factorial {
    # Initialize the result to 1
    result=1

    # Iterate from 1 to the number
    for i in $(seq 1 $1)
    do
        # Multiply the result by the current value of i
        result=$((result*i))
    done

    # Return the result
    echo $result
}

# Call the "factorial" function and print the result
echo "5! = $(factorial 5)"
```

Using Curl in Bash

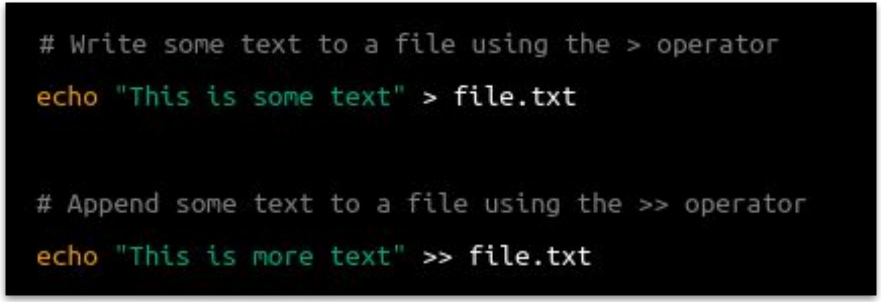


```bash
# Set the URL of the HTML file
url="https://example.com/page.html"

# Download the HTML file using curl
curl "$url" > page.html

# Extract the title of the page using grep and awk
title=$(grep -oP "(?<=<title>).*(?=</title>)" page.html | awk '{print $1}')

# Print the title to the terminal
echo "Title: $title"
```

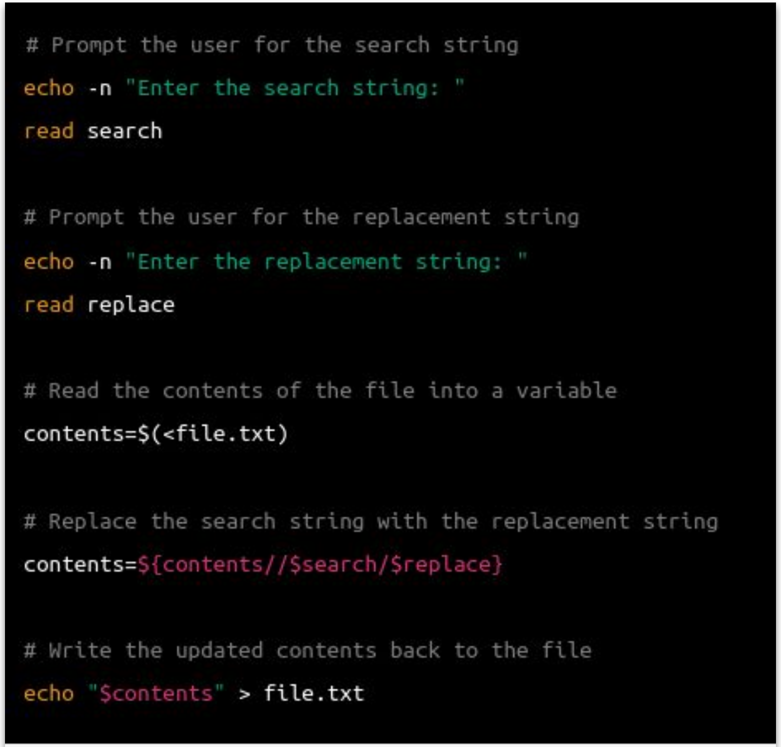
Input/output and File Manipulation



```bash
# Prompt the user for input
echo -n "Enter your name: "

# Read the input from the terminal
read name

# Print the input to the terminal
echo "Your name is: $name"
```



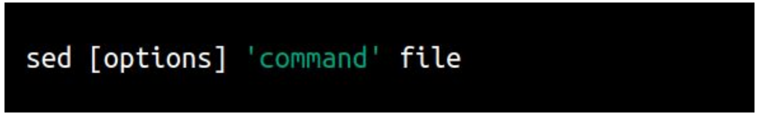
```bash
# Write some text to a file using the > operator
echo "This is some text" > file.txt

# Append some text to a file using the >> operator
echo "This is more text" >> file.txt
```



```bash
# Prompt the user for the search string
echo -n "Enter the search string: "
read search

# Prompt the user for the replacement string
echo -n "Enter the replacement string: "
read replace

# Read the contents of the file into a variable
contents=$(<file.txt)

# Replace the search string with the replacement string
contents=${contents//$search/$replace}

# Write the updated contents back to the file
echo "$contents" > file.txt
```

sed

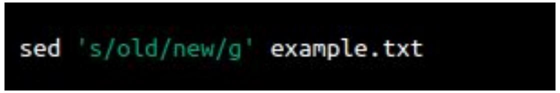
- sed is a powerful command-line tool for text manipulation in Bash
- The basic syntax for the sed command is as follows:



```
sed [options] 'command' file
```

- options: command line options
- command: text manipulation command
- file: target file where command will be performed

sed (Cont.)

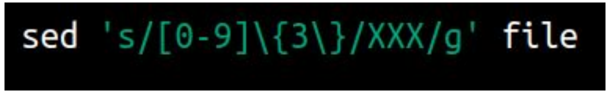


```
sed 's/old/new/g' example.txt
```

The above command will search for the word "old" in the file "example.txt" and replace it with the word "new".

The s is used for text substitution

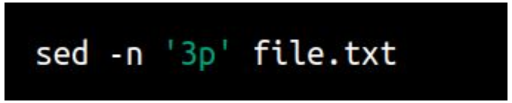
The g flag at the end of the command tells sed to make the replacement globally, meaning it will replace all occurrences of "old" in the file.



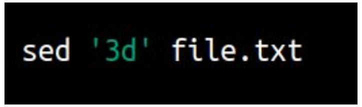
```
sed 's/[0-9]\{3\}/XXX/g' file
```

You can use regular expression to search in the file. The above command will replace 3 digits number in the give file

Sed can be used to work with the specific lines in the text and also can be used for advance text manipulation.

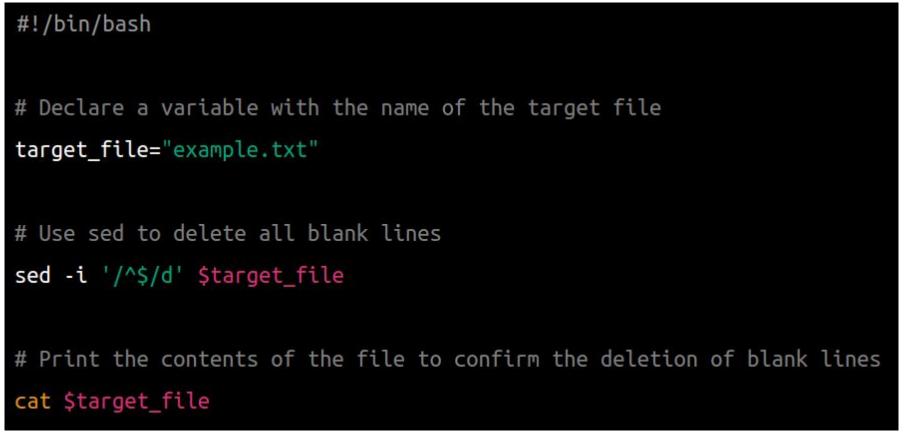


```
sed -n '3p' file.txt
```

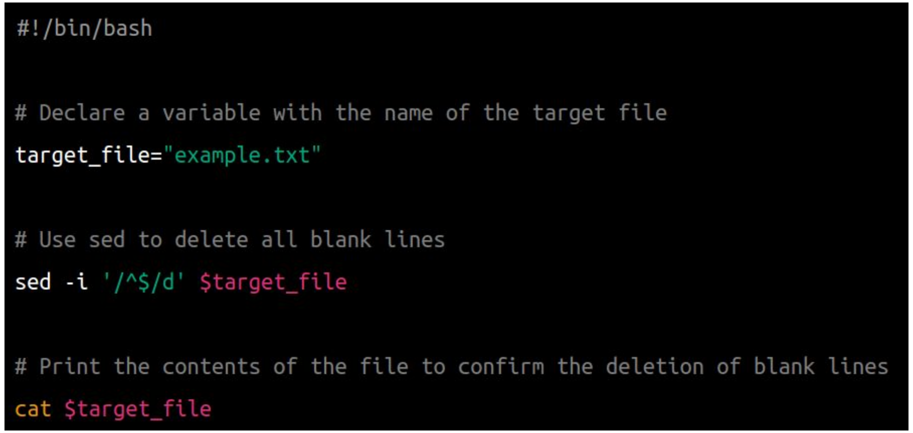


```
sed '3d' file.txt
```

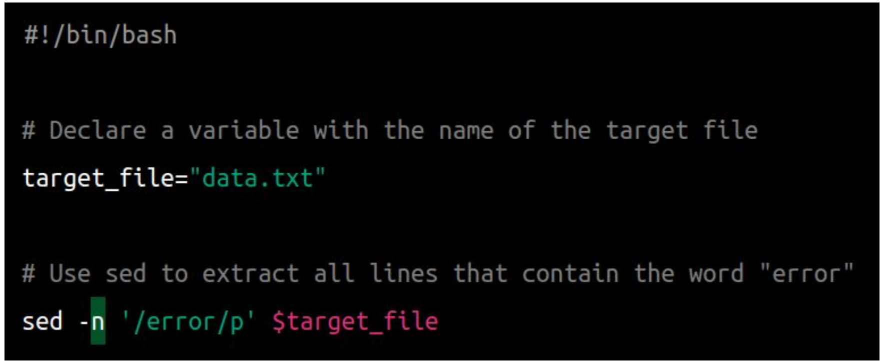
sed in Bash Script



```bash
#!/bin/bash

# Declare a variable with the name of the target file
target_file="example.txt"

# Use sed to delete all blank lines
sed -i '/^$/d' $target_file

# Print the contents of the file to confirm the deletion of blank lines
cat $target_file
```



```bash
#!/bin/bash

# Declare a variable with the name of the target file
target_file="example.txt"

# Use sed to delete all blank lines
sed -i '/^$/d' $target_file

# Print the contents of the file to confirm the deletion of blank lines
cat $target_file
```



```bash
#!/bin/bash

# Declare a variable with the name of the target file
target_file="data.txt"

# Use sed to extract all lines that contain the word "error"
sed -n '/error/p' $target_file
```

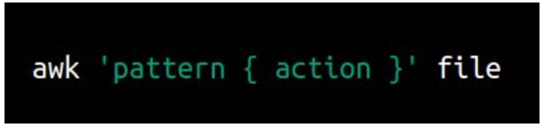
- This above script will extract all the lines that contain the word "error" from the file called "data.txt"
- The -n option is used to suppress the default output of sed
- If you remove -n the sed will also print the entire file
- The p flag is used to print the matched lines

awk

- awk is a powerful tool that can be used to manipulate text files
- commonly used to extract useful information from log files and CSV files and many more
- very useful in bash scripting and combining with other commands to achieve complex tasks

awk (cont.)

- The basic syntax to use the command:



```
awk 'pattern { action }' file
```

- pattern is an optional regular expression that is used to match against each line of the input file. If the pattern is not specified, the action will be performed on all lines of the input file
- action is a set of commands that are executed for each line that matches the pattern. The action is enclosed in curly braces {} and can include commands such as print, if-else statements, and variable assignments
- file is the input file that awk reads and processes

awk (cont.)

```
awk '{ print $2 }' file
```

```
awk '{ print $1, $3 }' file
```

```
awk '{ sum += $1 } END { print sum }' file
```

```
awk '{ if ($1 > 10) print $0 }' file
```

```
awk '{print toupper($0)}' file
```

```
awk '{ x = $1; printf("%d\n", x*x) }' file
```

```
awk '{ x = $1; if (x > 10) print $0 }' file
```

1. Print the second field of each line in file
2. Print the first and third fields of each line
3. Perform calculations on a specific field

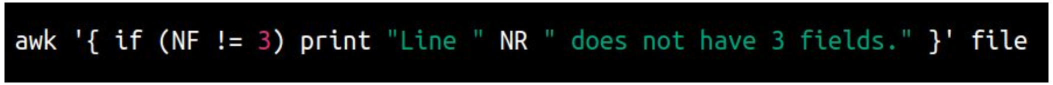
Try to understand for the rest of the examples…

awk (Cont.)

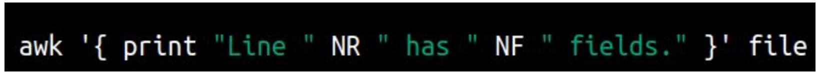
Here are some of the most commonly used built-in variables in awk:

1. $0 - refers to the entire line of input
2. $1, $2, $3, etc. - refers to the first, second, third, etc. field of the input, respectively.
3. The field separator (FS) is used to delimit the fields, by default it is a whitespace.
4. NF - refers to the number of fields in the current line
5. NR - refers to the current line number

There are more…



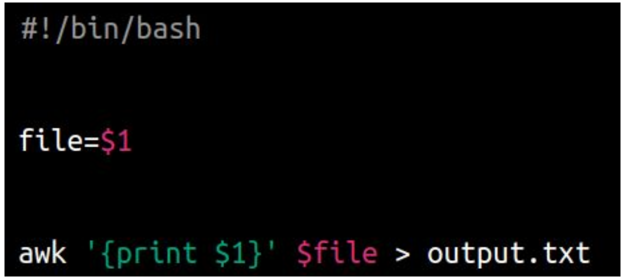
```
awk '{ if (NF != 3) print "Line " NR " does not have 3 fields." }' file
```



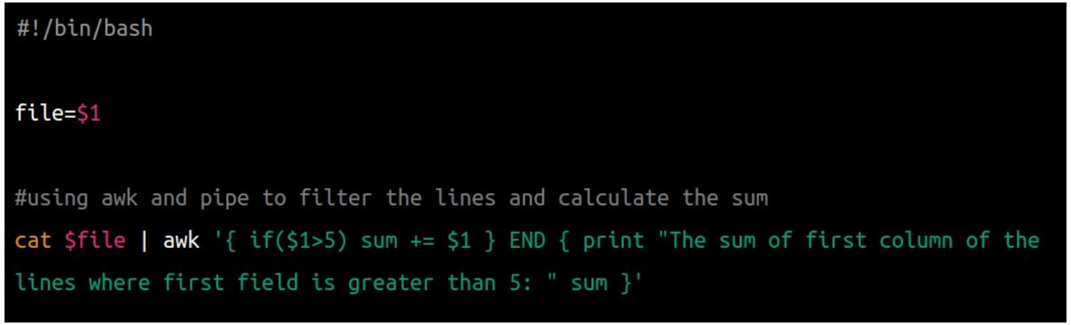
```
awk '{ print "Line " NR " has " NF " fields." }' file
```

```
awk '{ if (NF != 3) print "Line " NR " does not have 3 fields." }' file
```

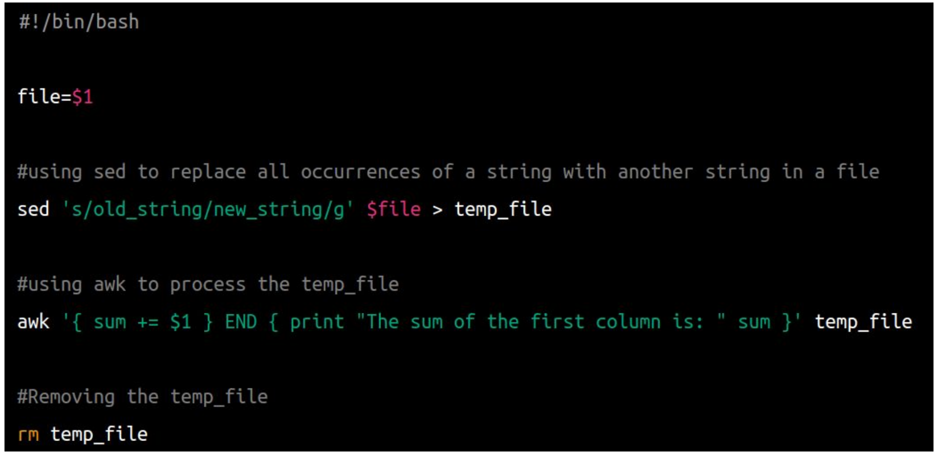
awk in Bash script



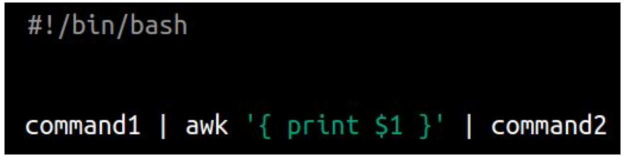
```
#!/bin/bash

file=$1

awk '{print $1}' $file > output.txt
```



```
#!/bin/bash

file=$1

#using awk and pipe to filter the lines and calculate the sum
cat $file | awk '{ if($1>5) sum += $1 } END { print "The sum of first column of the
lines where first field is greater than 5: " sum }'
```



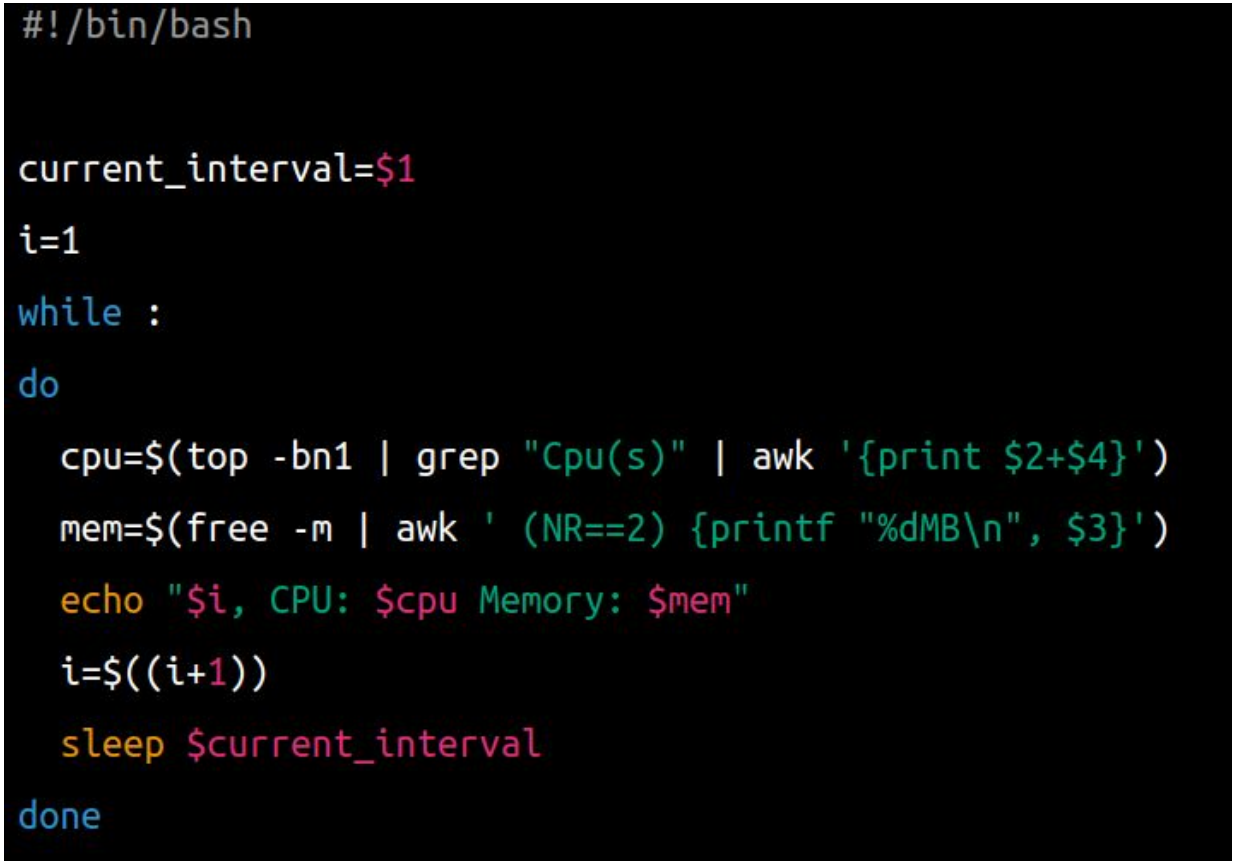
```
#!/bin/bash

file=$1

#using sed to replace all occurrences of a string with another string in a file
sed 's/old_string/new_string/g' $file > temp_file

#using awk to process the temp_file
awk '{ sum += $1 } END { print "The sum of the first column is: " sum }' temp_file

#Removing the temp_file
rm temp_file
```



```
#!/bin/bash

command1 | awk '{ print $1 }' | command2
```

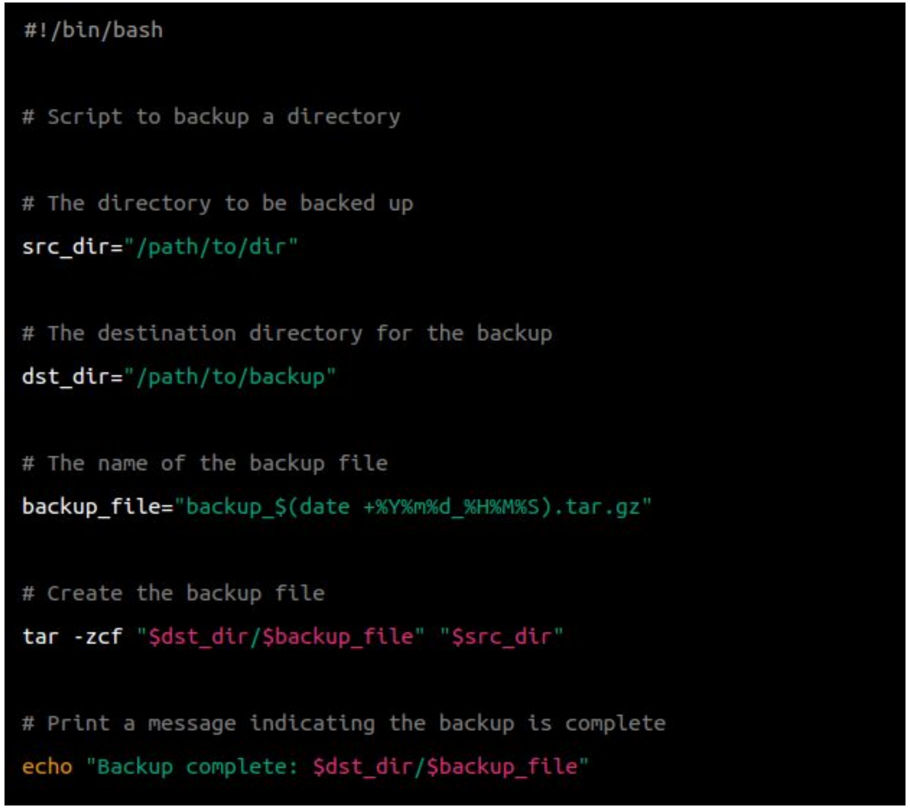
- Write a small script to check if “hello” exists in any line as second word in a text file.

Output the Script



```
#!/bin/bash


current_interval=$1

i=1

while :

do

  cpu=$(top -bn1 | grep "Cpu(s)" | awk '{print $2+$4}')

  mem=$(free -m | awk ' (NR==2) {printf "%dMB\n", $3}')

  echo "$i, CPU: $cpu Memory: $mem"

  i=$((i+1))

  sleep $current_interval

done
```

Output the Script



```
#!/bin/bash

# Script to backup a directory

# The directory to be backed up
src_dir="/path/to/dir"

# The destination directory for the backup
dst_dir="/path/to/backup"

# The name of the backup file
backup_file="backup_$(date +%Y%m%d_%H%M%S).tar.gz"

# Create the backup file
tar -zcf "$dst_dir/$backup_file" "$src_dir"

# Print a message indicating the backup is complete
echo "Backup complete: $dst_dir/$backup_file"
```

Debugging Bash Scripts

- Using the bash -n script.sh command: This will check the syntax of the script without executing.
- Using the -x option: -x option to the shebang line of your script, like so: #!/bin/bash -x. This will print each command to the terminal before it is executed, allowing you to see exactly what's happening at each step of the script.
- Using the set -x command: set -x at the beginning of your script, and set +x at the end of the script. This will enable and disable debugging for the entire script.
- Using the echo commands in the script
- Using the bash -v script.sh command: This will show each command before it's executed, same as -x option.

Use Cases of Bash Scripts

- Automating system administration tasks
- Automating application deployment
- Monitoring and logging
- Automating Build and CI/CD pipeline
- Automating cloud provisioning and infrastructure management
- Anything you can do through command in Linux can be automated through bash scripts!

Coming Next!

- Virtual Machines and Containers