

PERIYAR UNIVERSITY

(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3))

State University - NIRF Rank 56 - State Public University Rank 25

SALEM - 636 011

CENTRE FOR DISTANCE AND ONLINE EDUCATION

(CDOE)

MASTER OF COMPUTER APPLICATIONS

SEMESTER - I



CORE – I: LINUX AND SHELL PROGRAMMING

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

MCA 2024 admission onwards

Core Course – I

LINUX AND SHELL PROGRAMMING

Prepared by:

Centre for Distance and Online Education (CDOE)

Periyar University

Salem - 636011.

LIST OF CONTENTS

| UNIT | CONTENTS | PAGE |
|------|---|------|
| 1 | <p>Basic bash Shell Commands: Interacting with the shell- Traversing the file system-Listing files and directories-Managing files and directories-Viewing file contents.</p> <p>Basic Script Building: Using multiple commands-Creating a script file-Displaying messages- Using variables-Redirecting input and output-Pipes-Performing math-Exiting the script.</p> <p>Using Structured Commands: Working with the if-then statement-Nesting ifs-Understanding the test command-Testing compound conditions-Using double brackets and parentheses-Looking at case.</p> | 7 |
| 2 | <p>More Structured Commands: Looping with for statement-Iterating with the until statement- Using the while statement-Combining loops-Redirecting loop output.</p> <p>Handling User Input: Passing parameters-Tracking parameters-Being shifty-Working with options-Standardizing options-Getting user input.</p> <p>Script Control: Handling signals-Running scripts in the background-Forbidding hang-ups - Controlling a Job-Modifying script priority-Automating script execution.</p> | 45 |
| 3 | <p>Creating Functions: Basic script functions-Returning a value-Using variables in functions- Array and variable functions-Function recursion-Creating a library-Using functions on the command line.</p> <p>Writing Scripts for Graphical Desktops: Creating text menus-Building text window widgets- Adding X Window graphics.</p> <p>Introducing Sed and Gawk: Learning about the sed Editor-Getting introduced to the gawk Editor-Exploring sed Editor basics.</p> | 97 |
| 4 | <p>Regular Expressions: Defining regular expressions-Looking at the basics-Extending our patterns-Creating expressions.</p> <p>Advanced Sed: Using multiline commands-Understanding the hold space-Negating a command- Changing the flow-Replacing via a pattern-Using sed in scripts-Creating sed utilities.</p> <p>Advanced gawk: Reexamining gawk-Using variables in gawk-Using structured commands- Formatting the printing-Working with functions.</p> | 129 |
| 5 | <p>Working with Alternative Shells: Understanding the dash shell- Programming in the dash shell- Introducing the zsh shell-Writing scripts for zsh.</p> <p>Writing Simple Script Utilities: Automating backups-Managing user accounts-Watching disk space</p> | 165 |

| | | |
|--|---|--|
| | <p>Producing Scripts for Database, Web, and E-Mail: Writing database shell scripts-Using the Internet from your scripts-Emailing reports from scripts</p> <p>Using Python as a Bash Scripting Alternative: Technical requirements-Python Language- Hello World the Python way - Pythonic arguments-Supplying arguments-Counting arguments-Significant whitespace-Reading user input-Using Python to write to files-String manipulation.</p> | |
|--|---|--|

| UNIT | CONTENTS | PAGE |
|-------------|--|-------------|
| 1.1 | Basic bash Shell Commands | 7 |
| 1.2 | Basic Script | 14 |
| 1.3 | Using Structured Commands | 29 |
| 2.1 | More Structured Commands | 45 |
| 2.2 | Handling User Input | 57 |
| 2.3 | Script Control | 78 |
| 3.1 | Creating Functions | 97 |
| 3.2 | Writing Scripts for Graphical Desktops | 107 |
| 3.3 | Introducing Sed and Gawk | 112 |
| 4.1 | Regular Expressions | 129 |
| 4.2 | Advanced Sed | 140 |
| 4.3 | Advanced gawk. | 148 |
| 5.1 | Working with Alternative Shells | 165 |
| 5.2 | Writing Simple Script Utilities | 176 |
| 5.3 | Producing Scripts for Database, Web, and E-Mail | 188 |
| 5.4 | Using Python as a Bash Scripting Alternative | 202 |

Unit – I

Objectives :

- Understand what a shell is and the role of Bash in Unix-like operating systems.
- Learn and practice basic Bash commands and navigation techniques.
- Automate script execution and schedule tasks using cron jobs.

1.1 Basic bash Shell Commands

Starting the Shell:

- The default shell in many Linux distributions is the GNU bash shell.
- The shell is a program that provides interactive access to the Linux system and is typically started when a user logs in to a terminal.
- The shell used depends on the user's configuration, as specified in the `/etc/passwd` file. The last field in each entry of this file specifies the user's default shell program.

Using the Shell Prompt:

- After starting a terminal or logging into a Linux virtual console, you get access to the shell command-line interface (CLI) prompt.
- The default prompt symbol for the bash shell is the dollar sign (\$), which indicates that the shell is ready to accept your commands.
- The prompt can also display additional information, such as the current user's name and the system's name, which can be customized.
- When you enter a shell command at the prompt, you need to press the Enter key for the shell to execute the command.

Modifying the Shell Prompt:

- The shell prompt is not static and can be customized to suit your preferences.

Interacting with the shell

Interacting with the bash Manual:

- Most Linux distributions include an online manual for looking up information on shell commands and GNU utilities. The ``man`` command provides access to the manual pages stored on the Linux system. You can use it by entering ``man`` followed by a specific command name to access that utility's manual entry.
- Manual pages are displayed with a pager that allows you to navigate through them using spacebar, Enter key, or arrow keys.
- To exit the manual pages and return to the shell prompt, press the ``q`` key.
- The ``man`` command can be used to view manual pages about itself by typing ``man man``.

Sections in Manual Pages:

- The manual pages are divided into separate sections, each with a conventional naming standard. These sections provide information about a command's name, syntax, configuration, description, options, and more. Not all commands have all the listed sections, and some may have additional sections.

Searching for Commands:

- If you can't remember the command name, you can search the manual pages using keywords with the syntax ``man -k keyword``. For example, to find commands related to terminals, you can use ``man -k terminal``.

Section Areas:

- Manual pages are organized into section areas, each with an assigned number. The lowest numbered section is typically provided for a command. For example, (1) indicates executable programs or shell commands, while (7) is for overviews, conventions, and miscellaneous.

Additional Resources:

- In addition to `man` pages, there are also information pages called "info" pages. Most commands accept the `-help` or `--help` option for quick help. Several resources are available for reference, but detailed explanations may be needed for basic shell concepts.

Traversing the file system

Traversing Directories:

- The `cd` (change directory) command is used to navigate within the Linux filesystem. Its syntax is straightforward: `cd destination`. If no destination is specified, it takes you to your home directory.
- **Using Absolute Directory References:** Absolute directory references start from the root directory and are represented by a forward slash (/). For example, `cd /usr/bin` takes you to the `/usr/bin` directory.
- **Using Relative Directory References:** Relative references specify a destination relative to your current location and do not start with a forward slash (/). You can use directory names or special characters like `..` (parent directory). For instance, `cd Documents` moves to the `Documents` directory from your home directory.

Special Characters for Relative References:

- `.` represents the current directory.
- `..` represents the parent directory and is useful for moving up the directory

hierarchy.

- Remember to use absolute references when you're new to the command line and the Linux directory structure. As you become more familiar, you can switch to relative references.
- You can also display your current directory with the ``pwd`` command, ensuring you're in the right location before executing commands. Using relative references is especially useful within your home directory.

Listing files and directories

Listing Files and Directories:

- To view files on the system, use the ``ls`` (list) command. It can display basic or detailed information about files and directories. You can also filter listings based on names or patterns.

Displaying a Basic Listing:

- The basic ``ls`` command shows files and directories in your current directory. By default, it lists non-hidden directories alphabetically.
- Example:

```
``shell
```

```
$ ls
```

```
Desktop Downloads Music Pictures Templates Videos  
Documents examples.desktop my_script Public test_file
```

```
...
```

- You can use ``-F`` to flag directories with a slash and executables with an asterisk for easy identification.
- Hidden files (those starting with a dot) are not displayed by default. To show them, use ``-a``.
- The ``-R`` option displays files in subdirectories as well.

Displaying a Long Listing:

- For more detailed information about each file or directory, use the `-l` option with `ls`. It provides data like file type, permissions, owner, group, size, and modification time.

Filtering Listing Output:

- You can filter listings using text-matching strings and standard wildcard characters.
- `?` matches a single character, and `*` matches any number of characters.

- Example:

```
``shell
```

```
$ ls -l my_script
```

```
-rwxrw-r-- 1 christine christine 54 May 21 11:26 my_script
```

```
...
```

- File globbing, using brackets `[]`, allows for more advanced filtering. You can specify choices or character ranges.

- Example:

```
``shell
```

```
$ ls -l f[ai]ll
```

```
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fall
```

```
-rw-rw-r-- 1 christine christine 0 May 21 13:44 fill
```

```
...
```

- Use `!` to exclude specific patterns from the filter.
- File globbing is a powerful feature for searching files and can be used with other shell commands as well.

Managing files and directories

- The key points regarding managing files and directories, specifically creating files, copying files, handling file links, renaming files, creating directories, and deleting directories. Here's a concise summary:

Creating Files:

- You can create an empty file using the ``touch`` command, and you can specify the filename.
- The ``touch`` command can also be used to update the modification and access times of a file.

Copying Files:

- The ``cp`` command is used to copy files.
- The basic syntax of ``cp`` is ``cp source destination``.
- Use the ``-i`` option to prompt for confirmation when overwriting files.
- You can copy files to pre-existing directories or specify a different file name.

Handling File Links:

- Linux supports two types of file links: symbolic links and hard links.
- Symbolic links are separate files that point to another file's location.
- Hard links refer to the same physical file in different locations.
- Symbolic links can span different physical media, while hard links cannot.
- You can use the ``ln`` command to create links.

Renaming Files:

- To rename a file, use the ``mv`` command, specifying the old and new names.
- The ``mv`` command can also move files to different directories.
- It preserves the file's content and attributes.

Creating Directories:

- - Use the ``mkdir`` command to create new directories.

- - You can create multiple nested directories at once using the ``-p`` option.

Deleting Directories:

- To remove an empty directory, use the ``rmdir`` command.
- For directories with content, you can use ``rm -r`` to recursively remove them.
- Be cautious when using ``rm -r``, as it can't be undone.
- This summary provides a quick overview of the key concepts related to file and directory management in a Linux shell.

Viewing File Contents

Viewing the File Type

- Use the ``file`` command to determine the type of a file.
- It can identify text files, directories, symbolic links, scripts, and binary executable programs.

Viewing the Whole File

- To display the entire contents of a text file, use the ``cat`` command.
- You can use parameters like ``-n`` to number lines, ``-b`` to number lines with text, and ``-T`` to replace tabs with ``^I``.

Using the ``more`` Command

- The ``more`` command displays a text file one page at a time.
- Use the spacebar to navigate pages and ``q`` to quit.

Using the ``less`` Command

- ``less`` is an advanced version of ``more``, offering features like searching and navigation.

- It recognizes arrow keys and Page Up/Page Down keys.
- It can display a file's contents before reading the entire file.

Viewing Parts of a File

- Use the `tail` command to view the last lines of a file.
- Use `-n` to specify the number of lines, and `-f` to monitor the file in real-time.
- Use the `head` command to view the first lines of a file.
- Both commands help when you need to focus on specific parts of a file.

1.2 Basic Script Building

Using multiple commands

- The ability to run multiple commands and chain them together in a single step is a fundamental aspect of shell scripting. While using semicolons to separate commands is useful for small tasks or one-off operations, creating a shell script by combining multiple commands into a text file offers more convenience and flexibility.
- Here's how you can create a shell script to achieve the same result as the example you provided:
 - 1. Open a text editor (e.g., `nano`, `vim`, `gedit`, or any other text editor of your choice).
 - 2. Create a new text file and enter the following content:

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

```
# This is a simple shell script
```

```
# Run the 'date' command and display the current date and time date
```

```
# Run the 'who' command to display the list of logged-in users who
```

```
...
```

- In this script, `#!/bin/bash` is called a shebang, which specifies that the script should be executed using the Bash shell.

- 3. Save the file with a ".sh" extension, for example, "my_script.sh."
- 4. Make the script executable by running the following command in your terminal:

```
```bash
```

```
chmod +x my_script.sh
```

```
```
```

- Now, you can run the script by simply typing:

```
```bash
```

```
./my_script.sh
```

```
```
```

- The script will execute both the `date` and `who` commands, just like the example you provided. However, the advantage of using a script is that you don't need to manually type the commands every time you want to run them. You can edit and reuse the script as needed.

- In a more complex script, you can include variables, conditionals, loops, and functions to create powerful automation and process automation tasks. Shell scripting is a versatile way to perform various system tasks and automate repetitive actions.

Creating a script file

- Creating and running a shell script involves several steps, as you've described. Here's a summarized version for study material:

Create a Shell Script:

- Use a text editor to create a new file for your shell script. Start with a

shebang line to specify the shell you're using. For example:

```
``bash
#!/bin/b
ash
...

```

- You can add comments to describe the purpose of the script.

Add Commands:

- Enter your desired shell commands, one per line, in the script file. For example:

```
``bash

# This script displays the date and who's logged
ondate

who

...

```

Save the Script:

- Save the script with a ".sh" extension, for example, "my_script.sh."

Make it Executable:

- Use the `chmod` command to make the script executable:

```
``bash

chmod +x my_script.sh

...

```


Run the Script:

- To execute the script, run it using `./` followed by the script filename:

```
```bash
./my_script.sh
```
```

Permissions Issues:

- If you encounter a "Permission denied" error, change the file's permissions using `chmod` to give the owner execute permissions:

```
```bash
chmod u+x my_script.sh
```
```

Run the Script Again:

- Run the script again:

```
```bash
./my_script.sh
```
```

- shell script is now ready and can be executed whenever needed. Comments in the script help you understand the script's purpose and functionality, making it easier to maintain and modify the script in the future.

Displaying messages

- In shell scripting, you can use the `echo` command to display messages and output text to the console. Here's a summary of using `echo` for displaying messages in your shell scripts:

Basic Usage:

- You can use the `echo` command to display a simple text string. For example:

```
```bash
```

```
echo This is a test
```

```
```
```

- The output will be:

```
```
```

```
This is a test
```

```
```
```

Handling Quotes:

- When using quotes within your string, you should use the opposite type of quote to delimit the string. For example:

```
```bash
```

```
echo "Let's see if this'll work"
```

```
```
```

- The output will be:

```
```
```

```
Let's see if this'll work
```

```
```
```

- You can use either double quotes `"` or single quotes `'` to delimit your text strings.

Adding Messages to Shell Scripts:

- You can incorporate `echo` statements into your shell scripts to provide informative messages to users or to describe the script's actions. For example:

```
```bash
#!/bin/bas
h

This script displays the date and who's logged
onecho The time and date are:

date

echo "Let's see who's logged into the
system:"who
...

```

## Displaying Messages on the Same Line as Output:

- To display a message on the same line as command output, you can use the `-n` option with `echo`. For example:

```
```bash
echo -n "The time and date are: "
...

```

- This ensures that the echoed string and the command output are on the same line.
- The output will be:

```
...
The time and date are: Mon Feb 21 15:42:23 EST 2014
...

```

- Using the `echo` command allows you to provide information, status updates, or error messages within your shell scripts, making them more user-friendly and informative. It is a valuable tool for interaction between scripts and users.

Using variables

- Using variables in shell scripts is a powerful way to store and manipulate data. Here's a concise summary of how to use variables in your shell scripts:

Environment Variables:

- Environment variables are system-wide variables that store information like the username, home directory, and search paths. You can access them in your scripts using the ``$`` symbol. For example:

```
```bash
echo "User info for userid:
$USER"echo UID: $UID

echo HOME: $HOME
...`
```

### **User Variables:**

- You can create your own variables within your shell scripts. These user variables can hold values and are case-sensitive. Assign values to user variables using the equal sign `=` without any spaces. For example:

```
```bash
days=10
guest="Katie"
...`
```

Referencing Variables:

- When referencing the value of a variable, use the dollar sign ``$``. For example:

```
```bash
echo "The time and date are: $date"
...`
```

## Command Substitution:

- You can assign the output of a command to a variable using command substitution. This is done using backticks (`) or the `\$()` format. For example:

```
```bash
today=$(date +%y%m%d)
...`
```

- The `date` command's output is assigned to the variable `today`.

Subshells:

- Command substitution creates a subshell to run the enclosed command. Variables defined in your script are not available in subshells. Be aware of this when working with subshells.
- Using variables allows you to store, manipulate, and display data within your shell scripts, making them more versatile and useful for various tasks.

Redirecting input and output

- In shell scripting, you can redirect input and output to and from files using specific operators. Here's a concise summary of input and output redirection in shell scripts:

Output Redirection:

- The `>` symbol is used for output redirection. It allows you to save the output of a command to a file.
- For example, to save the output of the `date` command to a file named `output.txt`:

```
```bash
date > output.txt
...`
```

- If the file already exists, using `>` will overwrite its content.
- To append output to an existing file, use `>>`:

```
```bash
```

```
who >> output.txt
```

```
...
```

Input Redirection:

- The ``<`` symbol is used for input redirection. It allows you to provide input to a command from a file.
- For example, to count lines, words, and bytes in a file named `input.txt` using the `wc` command:

```
```bash
```

```
wc < input.txt
```

```
...
```

### Inline Input Redirection:

- The `<<` symbol allows you to provide data directly on the command line.
- You must specify a text marker (often referred to as a "here document") to indicate the beginning and end of the data.
- For example, counting lines, words, and bytes using `wc` with inline input redirection:

```
```bash
```

```
wc << EOF
```

```
Line 1
```

```
Line 2
```

```
Line
```

```
3
```

```
EOF
```

```
...
```

- The data entry continues until you enter the specified text marker ("EOF" in this

case).

- Output and input redirection are valuable techniques for manipulating data within your shell scripts, and they are often used for logging, processing large data sets, and interacting with external files.

Pipes

- In shell scripting, piping allows you to send the output of one command as input to another command. Piping is a powerful and efficient way to perform complex operations by chaining commands together. Here's a brief overview of piping in shell scripts:

Piping Symbol:

- The piping symbol is represented by `|`, which is often referred to as a "pipe."

Piping Syntax:

- To pipe the output of `command1` to the input of `command2`, use the following syntax:

```
```bash
```

```
command1 | command2
```

```
```
```

- The output from `command1` is immediately passed as input to `command2`.

Example:

- In the provided example, the output of the `rpm -qa` command (which lists installed packages) is piped to the `sort` command to sort the list alphabetically:

```
```bash
```

```
rpm -qa | sort
```

```
```
```

- The sorted list is displayed in real-time, with no intermediate files or buffers.

Chaining Pipes:

- You can chain multiple commands together using pipes. For example:

```
```bash
```

```
command1 | command2 | command3
```

```
```
```


- In this case, the output from `command1` is passed to `command2`, and the output from `command2` is passed to `command3`.

Pausing Output:

- When the output is too long to read at once, you can use text-paging commands like
`more` to pause and read the data screen by screen:

```
```bash
command1 | command2 | more
``
```

- This allows you to control the flow of data for easier reading.

## Saving Output to a File:

- You can combine piping with output redirection to save the final output to a file.  
For example:

```
```bash
command1 | command2 > output.txt
``
```

- This sends the output of `command1 | command2` to a file called `output.txt`.
- Piping is a fundamental concept in shell scripting and is widely used to process data and execute complex tasks efficiently.

Performing math

- Performing mathematical operations in shell scripts can be accomplished using various methods. In the provided text, three different methods are explained: ``expr``, using square brackets, and using the ``bc`` (bash calculator) command for floating-point calculations. Here's a summary of each:

``expr`` Command:

- The ``expr`` command is a basic way to perform integer arithmetic in shell scripts.
- It recognizes various mathematical and string operators.
- To use it, you need to escape characters that may be misinterpreted by the shell (e.g., ``expr 5 * 2``).

Using Square Brackets:

- In bash, you can use square brackets to perform integer arithmetic (``$[operation]``).
- This method simplifies integer arithmetic and doesn't require escaping operators.
- It's suitable for simple calculations but limited to integer arithmetic.

``bc`` (Bash Calculator) for Floating-Point Arithmetic:

- The ``bc`` command is a full-featured calculator that supports floating-point arithmetic.
- You can access ``bc`` from the command line and set the scale (decimal places) for results.
- The ``bc`` command can be used within shell scripts for more complex arithmetic operations.
- Here's how to use ``bc`` in shell scripts to perform floating-point arithmetic:

```
``bash
#!/bin/bash
var1=20
var2=3.14159

var3=$(bc << EOF
scale=4
```

```
result = $var1 * $var2
result
EOF
)
echo The final result is $var3
...

```

- This script sets variables ``var1`` and ``var2`` and then uses ``bc`` with inline input redirection to calculate ``var1 * var2``. The result is stored in the ``var3`` variable, which is then displayed. ``bc`` allows you to perform more complex calculations and supports floating-point numbers.
- These three methods provide different options for performing mathematical operations in shell scripts, depending on your specific needs, whether it's simple integer arithmetic, more advanced integer operations, or floating-point arithmetic.

Exiting the script

- In shell scripting, you can gracefully exit a script using the ``exit`` command and specify an exit status code to indicate the script's completion status. The exit status is an integer value between 0 and 255 that is returned by the script to the calling environment. Here's a summary of how to use the ``exit`` command:

Checking the Exit Status:

- You can check the exit status of a command immediately after it has executed using the special variable ``${}``. A value of 0 typically indicates successful completion, while non-zero values often indicate errors.
- Common exit status codes and their meanings are given in Table 11-2 in the provided text.

Using the ``exit`` Command:

- The ``exit`` command allows you to set the exit status explicitly when ending a script.

- You can specify the desired exit status as a parameter to the `exit` command.
- This is useful for indicating the outcome of your script and can be helpful when your script is used in automated processes.
- Here's how to use the `exit` command in a script:

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

# Testing the exit status
var1=10

var2=30 var3=${var1
+ $var2]

echo The answer is $var3
exit 5

...`
```

- In this example, the script will exit with an exit status of 5. When you check the exit status using `echo \$?`, you'll see the value is 5.
- Keep in mind that exit status codes should typically be in the 0 to 255 range. If you specify a value that exceeds this range, the shell will calculate the modulo (remainder) of the value. For example, if you specify an exit status of 300, the actual exit status will be 44 (300 modulo 256). It's a good practice to use meaningful exit status codes to indicate success or specific types of errors in your scripts, making it easier to handle script outcomes in automated workflows.

1.3 Using Structured Commands

Working with the if-then statement

- The `if-then` statement in a Bash script allows you to execute a block of commands conditionally based on the exit status of a preceding command. here's how it works:
- The `if` statement begins the conditional block and specifies the command to run.

- The `then` keyword marks the beginning of the commands to be executed if the preceding command has a zero exit status (indicating success).
- The `fi` statement marks the end of the `if-then` block.
- The `if` statement doesn't evaluate whether a condition is true or false, as it might in some other programming languages. Instead, it checks the exit status of the command. If the exit status is zero (success), the commands in the `then` block are executed. If the exit status is non-zero (failure), the commands in the `then` block are skipped.
- In your script, you can have multiple commands within the `then` block. They are treated as a block of code and are executed together if the exit status of the initial command in the `if` statement is zero.
- Additionally, you can use the `else` and `elif` (else if) statements to provide alternative actions when the initial command has a non-zero exit status. Here's the basic structure:

```

```bash

if command1; then

 # commands to run if command1
 succeedselse

 # commands to run if command1 fails

fi
```

```

- Or, with `elif`:

```

```bash

if command1; then

 # commands to run if command1
 succeedselif command2; then

 # commands to run if command1 fails and command2
 succeedselse

 # commands to run if both command1 and command2 fail

```

fi

...

- These statements allow you to add more flexibility to your script by specifying different actions for various conditions.

## **Nesting ifs**

- Nesting if-then statements in shell scripting is a common practice to handle multiple conditional situations. However, as the example you provided shows, it can make the code hard to read and follow, especially when dealing with multiple conditions. To address this, you can use ``elif`` statements to create a cleaner and more organized structure for handling different conditions.
- Here's a summary of what the example demonstrates:
  1. The first script checks if a user exists in ``/etc/passwd``, and if not, it checks for the existence of a directory using nested if-then statements.
  2. The second script improves the code by using ``elif`` statements instead of nesting if-then statements, resulting in cleaner and more readable code.
  3. The third script further enhances the code by adding an ``else`` block within the ``elif`` block to handle the case where the user doesn't exist and doesn't have a directory.
- The use of ``elif`` statements makes it easier to handle multiple conditions and provides a more structured and readable script. However, as mentioned in the text, if you have a large number of conditions, you might want to consider using the ``case`` command for better code organization.
- The ``case`` command allows you to match a variable against multiple patterns, making it a more suitable choice for handling complex conditional logic. It's especially helpful when you have a long list of possible values to compare. Here's a simplified example of how you can use the ``case`` command to achieve the same outcome as the third script:

```

```bash
#!/bin/bash

# Testing the case command
testuser=NoSuchUser

case $testuser in

$(grep "$testuser" /etc/passwd)

    echo "The user $testuser exists on this system."

    ;;

$(ls -d /home/$testuser 2>/dev/null)

    echo "The user $testuser does not exist on this system."
    echo "However, $testuser has a directory."

    ;;

*)

    echo "The user $testuser does not exist on this system."
    echo "And, $testuser does not have a directory."

    ;;

esac
```

```

- In this script, the `case` command compares the value of `testuser` against different patterns. The `\*)` at the end acts as a catch-all for cases that don't match the previous patterns. This approach can make your code more maintainable and easier to follow when dealing with multiple conditions.
- If you have a large number of conditions to check, the `case` command is a better choice than nesting numerous if-then statements or `elif` blocks.

## **Understanding the test command**

- In Bash scripting, you can use conditional statements like `if` to evaluate various conditions. To check conditions other than exit status codes of commands, you

can use the ``test`` command within ``if-then`` statements.

- The ``test`` command allows you to assess different conditions. If the condition is true, the ``test`` command exits with a status code of 0, making the ``if-then`` statement behave like those in other programming languages. If the condition is false, the ``test`` command exits with a non-zero status code, causing the ``if-then`` statement to exit.
- 
- Here's a basic format:

```
``bash
if test condition
then
 # commands
fi
...

```

- You can also use square brackets `[ ]` to define test conditions without the ``test`` command:

```
``bash
if [condition]
then
 # commands
fi
...

```

- The ``test`` and `test` conditions can evaluate numeric comparisons, string comparisons, and file comparisons. Numeric comparisons involve operators like ``-eq``, ``-ge``, ``-gt``, ``-le``, ``-lt``, and ``-ne``. String comparisons include ``=``, ``!=``, ``<``, ``>``,



``-n``, and ``-z``. File comparisons check for file attributes, such as ``-d``, ``-e``, ``-f``, ``-r``,

``-s``, ``-w``, ``-x``, ``-O``, ``-G``, ``-nt``, and ``-ot``.

- These conditions help you make decisions and control the flow of your Bash scripts.

## **Testing compound conditions**

- The provided shell script demonstrates the use of the AND Boolean operator (`&&`) to combine two conditions in an `if-then`` statement. Here's a summary of what the script does:

1. It checks if the user's home directory (``${HOME}``) exists.
2. It checks if there's a file named "testing" in the user's home directory (``${HOME}``)

and

- whether the user has write permissions for that file.
- If both of these conditions are met, the script will print "The file exists and you can write to it." Otherwise, it will print "I cannot write to the file."
- To summarize, the script uses the AND Boolean operator to ensure that both conditions must be TRUE for the `then`` section to execute. If either of the conditions is FALSE, the `else`` section will be executed.
- You can adapt this script as an example of using compound tests with the AND operator. It's a common practice to check multiple conditions before performing certain actions in shell scripts.

## **Using double brackets and parentheses**

- The provided information explains two advanced features that you can use in `if-then`` statements in Bash:

## Double Parentheses for Mathematical Expressions:

- You can use double parentheses `((...))` to incorporate advanced mathematical expressions in your comparisons.
- Double parentheses allow for a wide range of mathematical operators and expressions, including post-increment, post-decrement, pre-increment, pre-decrement, logical negation, exponentiation, bitwise shifts, bitwise Boolean operations, logical AND, and logical OR.
- You can use double parentheses in an `if` statement to perform complex mathematical comparisons and assignments. For example:

```
```bash
# Example of using double parentheses for mathematical comparisons
val1=10
if (( $val1 ** 2 > 90 ))
then
    (( val2 = $val1 ** 2 ))
    echo "The square of $val1 is $val2"
fi
```
```

## Double Brackets for Advanced String Handling and Pattern Matching:

- Double brackets `[...]` provide advanced features for string comparisons, including pattern matching.
- You can use double brackets for string comparisons and apply regular expressions to match strings.
- In the example provided, `[ \$USER == r\* ]` is used to match the `USER` environment variable to see if it starts with the letter "r." If it does, the `then` section

isexecuted.

```
```bash
```

```
# Example of using double brackets for string comparisons with pattern  
matchingif [[ $USER == r* ]]
```

```
then
```

```
    echo "Hello $USER"
```

```
else
```

```
    echo "Sorry, I do not know you"
```

```
fi
```

```
```
```

- It's important to note that while double brackets work well in the Bash shell, not all shells support this feature. Double brackets are particularly useful when you need to perform complex string comparisons and pattern matching in your scripts.

## **Looking at case**

- The `case` command in Bash is a more concise and cleaner way to handle multiple conditions for a single variable, as opposed to writing a lengthy `if-then-else` statement. Here's a summary of how to use the `case` command:
- Original `if-then-else` example:

```
```bash
```

```
if [ $USER = "rich" ]
```

```
then
```

```
    echo "Welcome $USER"
```

```
    echo "Please enjoy your visit"
```

```

elif [ $USER = "barbara" ]
then

    echo "Welcome $USER"
    echo "Please enjoy your visit"
elif [ $USER = "testing" ]

then

    echo "Special testing account"

elif [ $USER = "jessica" ]
then

    echo "Do not forget to logout when you're done"
else

    echo "Sorry, you are not allowed here"
fi
...

```

- Using the `case` command for the same purpose:

```

``bash

case $USER in
rich | barbara)

    echo "Welcome, $USER"

    echo "Please enjoy your visit";;
testing)

    echo "Special testing account";;
jessica)

    echo "Do not forget to log off when you're done";;
*)

    echo "Sorry, you are not allowed here";;
esac

```

...

- Key points about the `case` command:
 - It allows you to compare a single variable against different patterns.
 - You can list multiple patterns on the same line using the `|` operator to separate them.
 - The `*` pattern serves as a catch-all for values that don't match any of the specified patterns.
 - For each pattern that matches the variable, you can specify the commands to execute using `;;`.
- The `case` command provides a more efficient and readable way to handle multiple conditions for the same variable, making your scripts cleaner and more maintainable.

Unit Summary

Structured commands allow you to alter the normal flow of shell script execution. The most basic structured command is the if-then statement. This statement provides a command evaluation and performs other commands based on the evaluated command's output.

You can expand the if-then statement to include a set of commands the bash shell executes if the specified command fails as well. The if-then-else statement executes commands only if the command being evaluated returns a non-zero exit status code.

You can also link if-then-else statements together, using the `elif` statement. The `elif` is equivalent to using an else if statement, providing for additional checking of whether the original command that was evaluated failed.

In most scripts, instead of evaluating a command, you'll want to evaluate a condition, such as a numeric value, the contents of a string, or the status of a file or directory. The `test` command provides an easy way for you to evaluate all these conditions. If the condition evaluates to a TRUE condition, the `test` command produces a zero exit status code for the if-then statement. If the condition evaluates to a FALSE condition, the `test` command produces a non-zero exit status code for the if-then statement.

The square bracket is a special bash command that is a synonym for the test command. You can enclose a test condition in square brackets in the if-then statement to test for numeric, string, and file conditions.

The double parentheses command provides advanced mathematical evaluations using additional operators. The double square bracket command allows you to perform advanced string pattern-matching evaluations.

Let us sum up:

The GNU bash shell is a program that provides interactive access to the Linux system. It runs as a regular program and is normally started whenever a user logs in to a terminal.

The rmdir has no -i option to ask if you want to remove the directory. This is one reason it is helpful that rmdir removes only empty directories. You can also use the rm command on entire non-empty directories.

Check your progress

PART -A

1. Which command is used to change the current directory in a bash shell?*

- A) cd B) ls C) mv D) pwd

2. What does the command pwd do in a bash shell?*

- A) Changes the password B) Prints the current working directory
C) Lists the files in a directory D) Deletes a file

3. Which command lists all files, including hidden ones, in a directory?*

- A) ls B) ls -l C) ls -a D) ls -h

4. Which command is used to create a new directory in bash?*

- A) mkdir B) rmdir C) touch D) rm

5. Which command is used to display the contents of a file in the terminal?*

- A) cat B) ls C) rm D) cp

6. Which symbol is used to separate multiple commands on a single line in bash?*

- A) ; B) & C) | D) #

7. What is the correct file extension for a bash script?*

- A) .sh B) .bash C) .script D) .txt

8. Which command is used to display a message in a bash script?*

- A) echo B) print C) show D) display

9. How do you assign a value to a variable in bash?*

- A) variable = value B) variable=value C) \$variable = value D) var
value

10. Which symbol is used to redirect the output of a command to a file?*

- A) > B) < C) | D) &

11. What is the purpose of the pipe (|) command in bash?*

- A) To run commands in the background B) To combine two files
C) To send the output of one command as input to another command
D) To terminate a process

12. Which command is used to perform arithmetic operations in a bash script?*
- A) math B) calc C) expr D) compute
13. Which command is used to exit a script in bash?*
- A) exit B) quit C) stop D) end
14. What is the basic syntax of an if-then statement in bash?*
- A) if [condition]; then ... fi B) if (condition) { ... }
C) if [condition] { ... } endif D) if (condition); then ... done
15. How do you nest if statements in bash?*
- A) By using elif B) By using multiple if statements
C) By using case D) By using switch
16. Which command is used for evaluating conditions in bash scripts?*
- A) test B) check C) eval D) condition
17. Which operator is used to test compound conditions in bash?*
- A) && B) || C) both A and B D) either A or B
18. What is the difference between single [and double [[brackets in bash?*
- A) There is no difference.
B) Double brackets provide additional functionality.
C) Single brackets are used for strings only.
D) Double brackets are used for arithmetic only.
19. Which command is used to perform pattern matching in bash?*
- A) match B) case C) switch D) pattern

Here are the answers:

1. A) 2. B) 3. C) 4. A) 5. A) 6. A) 7. A) 8. A) 9. B) 10. A) 11. C) 12. C) 13. A) 14. A) 15. A) 16. A) 17. C) 18. B) 19. B)

Self Assessment Questions

1. Write difference between Soft and Hard links ?
2. Difference between Linux and Windows
3. Important feature of Linux OS
4. How to traverse the file system in Linux ?
5. Compare local variable and Global variable.
6. Define wildcard character with examples.
7. Write any 10 shell commands with example.
8. Explain Redirecting Input and Output with example.
9. Explain floating point solution with example.
10. Brief explain if statements with examples.

Glossary

1. `ls`

- **Description**: Lists the contents of a directory.

- **Usage**: `ls [options] [directory]`

- **Options**: `-l` (long format), `-a` (include hidden files), `-h` (human-readable sizes).

2. `cd`

- **Description**: Changes the current directory.

- **Usage**: `cd [directory]`

3. `pwd`

- **Description**: Prints the current working directory.

- **Usage**: `pwd`

4. `mkdir`

- **Description**: Creates a new directory.

- **Usage**: `mkdir [directory]`

5. `rmdir`

- **Description**: Removes an empty directory.

- **Usage**: `rmdir [directory]`

6. `rm`

- **Description**: Removes files or directories.

- **Usage**: `rm [options] [file/directory]`

- **Options**: `-r` (recursive, for directories), `-f` (force).

7. `cp`

- **Description**: Copies files or directories.

- **Usage**: `cp [options] [source] [destination]`

- **Options**: `-r` (recursive), `-i` (interactive).

8. `mv`

- **Description**: Moves or renames files or directories.

- **Usage**: `mv [source] [destination]`

9. `touch`

- **Description**: Creates an empty file or updates the timestamp of an existing file.

- **Usage**: `touch [file]`

10. `cat`

- **Description**: Concatenates and displays the content of files.

- **Usage**: `cat [file]`

11. `more`

- **Description**: Views the content of a file one page at a time.

- **Usage**: `more [file]`

12. `less`

- **Description**: Views the content of a file with backward movement capability.
- **Usage**: `less [file]`

13. `head`

- **Description**: Displays the first few lines of a file.
- **Usage**: `head [file]`

14. `tail`

- **Description**: Displays the last few lines of a file.
- **Usage**: `tail [file]`

15. `echo`

- **Description**: Displays a line of text or the value of a variable.
- **Usage**: `echo [text]`

16. `grep`

- **Description**: Searches for a pattern in files.
- **Usage**: `grep [options] [pattern] [file]`
- **Options**: `-i` (ignore case), `-r` (recursive), `-v` (invert match).

17. `find`

- **Description**: Searches for files and directories.
- **Usage**: `find [path] [options] [expression]`

18. `chmod`

- **Description**: Changes file permissions.
- **Usage**: `chmod [permissions] [file]`

19. `chown`

- **Description**: Changes file owner and group.
- **Usage**: `chown [owner][:group] [file]`

20. `df`

- **Description**: Displays disk space usage.
- **Usage**: `df [options]`
- **Options**: `-h` (human-readable).

Open-source e-content links:

<https://www.techtarget.com/searchdatacenter/definition/bash-Bourne-Again-Shell>

<https://opensource.com/resources/what-bash>

books

1. Pro Bash Programming: Scripting the Linux Shell by Chris F.A. Johnson and Jayant Kumar
2. "The Linux Command Line: A Complete Introduction" by William E. Shotts Jr.

UNIT – II

❖ Objective:

- Enhance the script's readability and maintainability by organizing commands and functions logically.
- Improve the script's interaction with users by effectively managing input and providing clear instructions and feedback.
- Manage the execution flow of the script efficiently, allowing for better control and error handling.

2.1 More Structured Commands

Looping with For Statement :

Iterating through a **series of commands** is a common programming practice. Often, you need to repeat a set of commands **until a specific condition** has been met, such as processing all the files in a directory., all the

The bash shell provides the **for command** to allow you to create a loop that iterates through a series of values. Each iteration performs a defined set of commands using one of the values in the series. Here's the basic format of the bash shell for command:

```
for var in list  
  
do  
  
commands  
  
done
```

You supply the **series of values used in the iterations in the list parameter**. You can specify the values in the list in several ways.

Reading values in a list :

The most basic use of the for command is to iterate through a list of values defined within the for command itself:

```
$ cat test1  
#!/bin/bash  
# basic for command  
for test in Akshaya Dhanush Gokul Gopika Gowri Gowtham  
do  
    echo The next state is $test  
done  
$ ./test1  
The next state is Akshaya  
The next state is Dhanush  
The next state is Gokul  
The next state is Gopika  
The next state is Gowri  
The next state is Gowtham  
$
```

Reading complex values in a list

There are times when you run into data that causes problems. Here's a classic example of what can cause problems for shell script programmers:

```
$ cat badtest1  
  
#!/bin/bash  
  
# another example of how not to use the for command
```

```
for test in I don't know if this'll work
```

```
do
```

```
    echo "word:$test"
```

```
done
```

```
$ ./badtest1
```

```
word:I
```

```
word:dont know if thisll
```

```
word:work
```

Reading complex values in a list :

- ▶ The shell saw the **single quotation marks** within the list values and attempted to use them to define a single data value, and it really messed things up in the process.
- ▶ You have two ways to solve this problem:
- ▶ **Use the escape character (the backslash) to escape the single quotation mark.**
- ▶ **Use double quotation marks to define the values that use single quotation marks.**
- ▶ Neither solution is all that fantastic, but each one helps solve the problem:
- ▶ **\$ cat test2**
- ▶ **#!/bin/bash**
- ▶ **# another example of how not to use the for command**
- ▶ **for test in I don't know if "this'll" work**
- ▶ **do**
- ▶ **echo "word:\$test"**

▶ **done**

Output

▶ **\$./test2**

▶ **word:l**

▶ **word:don't**

▶ **word:know**

▶ **word:if**

▶ **word:this'll**

▶ **word:work**

▶ **\$**

Reading a list from a variable :

Often what happens in a shell script is that you accumulate **a list of values stored in a variable** and then need to iterate through the list. You can do this using the `for` command as well:

▶ **\$ cat test4**

▶ **#!/bin/bash**

▶ **# using a variable to hold the list**

▶ **list="Salem Erode Namakkal Dharmapuri "**

▶ **list=\$list " Kallakurichi"**

▶ **for state in \$list**

▶ **do**

▶ **echo "Have you ever visited \$state?"**

▶ **done**

Output

\$./test4

Have you ever visited Salem?

Have you ever visited Erode?

Have you ever visited Namakkal?

Have you ever visited Dharmapuri?

Have you ever visited Kallakurichi?

\$

- ▶ The \$list variable contains the standard text list of values to use for the iterations.
- ▶ Notice that the code also uses another assignment statement to **add (or concatenate) an item** to the existing list contained in the \$list variable.
- ▶ This is a common method for **adding text to the end of an existing text string stored in a variable.**

Reading values from a command :

- ▶ \$ cat test5
- ▶ #!/bin/bash
- ▶ # reading values from a file
- ▶ file="states"
- ▶ for state in \$(cat \$file) do
- ▶ echo "Visit beautiful \$state"
- ▶ done
- ▶ \$ cat states
- ▶ Alabama
- ▶ Alaska
- ▶ Arizona
- ▶ Arkansas

This example uses the cat command in the command substitution **to display the contents of the file states**. Notice that the states file includes each state on **a separate line, not separated by spaces**. The for command still iterates through **the output of the cat command one line at a time**, assuming that each state is **on a separate line**.

- ▶ Colorado
- ▶ Connecticut
- ▶ Delaware
- ▶ Florida
- ▶ Georgia
- ▶ \$./test5
- ▶ Visit beautiful Alabama
- ▶ Visit beautiful Alaska
- ▶ Visit beautiful Arizona
- ▶ Visit beautiful Arkansas
- ▶ Visit beautiful Colorado
- ▶ Visit beautiful Connecticut
- ▶ Visit beautiful Delaware
- ▶ Visit beautiful Florida
- ▶ Visit beautiful Georgia
- ▶ \$

Changing the field separator :

- ▶ The cause of this problem is the **special environment variable IFS**, called the *internal field separator*.
- ▶ The IFS environment variable defines a list of characters the bash shell uses as **field separators**.
- ▶ By default, the bash shell considers the following characters as field separators:
 - ▶ A space
 - ▶ A tab
 - ▶ A newline
 - ▶ Eg. IFS=\$'\n'

- ▶ `IFS=$'\n':;`
- ▶ This assignment uses the **newline, colon, semicolon, and double quotation mark** characters as field separators.
- ▶ **Reading a directory using wildcards**
- ▶ Finally, you can use the `for` command to automatically iterate through a directory of files. To do this, you must **use a wildcard character** in the file or pathname. This forces the shell to use *file globbing*.
- ▶ `$ cat test6`
- ▶ `# iterate through all the files in a directory`
- ▶ `for file in /home/rich/test/* do`
- ▶ `if [-d "$file"] then`
- ▶ `echo "$file is a directory"`
- ▶ `elif [-f "$file"] then`
- ▶ `echo "$file is a file"`
- ▶ `fi done`

output `$./test6`

`/home/rich/test/dir1 is a directory`

`/home/rich/test/myprog.c is a file`

`/home/rich/test/myprog is a file`

`/home/rich/test/myscript is a file`

`/home/rich/test/newdir is a directory`

`/home/rich/test/testdir is a directory`

`/home/rich/test/testing is a file`

`/home/rich/test/testprog is a file`

`/home/rich/test/testprog.c is a file`

\$ The for command iterates through the results of the `/home/rich/test/*` listing.

The while Command :

- The while command is somewhat of a cross between the if-then statement and the for loop.
- The while command allows you to define a command to test and then loop through a set of commands for as long as the defined test command returns a zero exit status.
- It tests the test command at the start of each iteration.
- When the test command returns a non- zero exit status, the while command stops executing the set of commands.

Basic while format

Here's the format of the while command:

```
while test command
```

```
do
```

```
other commands
```

```
done
```

Example

The most common use of the test command is to use brackets to check a value of a shell variable that's used in the loop commands:

```
$ cat test10
```

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

```
# while command test
```

```
var1=10
```

```
while [ $var1 -gt 0 ] do
```

```
echo $var1
```

```
var1=$(( $var1 - 1 ]) done
```

\$./test10

The until Command

- ▶ The until command works in exactly the opposite way from the while command. The until command requires that you specify a test command that normally produces a non-zero exit status. As long as the exit status of the test command is non-zero, the bash shell executes the commands listed in the loop. When the test command returns a zero exit status, the loop stops.
- ▶ **As you would expect, the format of the until command is:**

until test *commands*

do

other commands

done

- ▶ Similar to the while command, you can have more than one *test command* in the until command statement. Only the exit status of the last command determines if the bash shell executes the *other commands* defined.

Example

- ▶ **The following is an example of using the until command:**

```
$ cat test12
```

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

```
# using the until command
```

```
var1=100
```

```
until [ $var1 -eq 0 ] do
```

```
echo $var1
```

```
var1=$(( $var1 - 25 ]) done
```

```
$ ./test12
```

Nesting Loops

- ▶ A loop statement can use any other type of command within the loop, including other loop commands. This is called a *nested loop*.
- ▶ Care should be taken when using nested loops, because you're performing an iteration within an iteration, which multiplies the number of times commands are being run.
- ▶ **Here's a simple example of nesting a for loop inside another for loop:**

```
$ $ cat test14  
  
#!/bin/bash  
  
# nesting for loops  
  
for (( a = 1; a <= 3; a++ )) do  
    echo "Starting loop $a:"  
    for (( b = 1; b <= 3; b++ )) do  
        echo "    Inside loop: $b" done  
    done  
done
```

Output

```
$ ./test14  
  
Starting loop 1:  
    Inside loop: 1  
    Inside loop: 2  
    Inside loop: 3  
Starting loop 2:  
    Inside loop: 1  
    Inside loop: 2  
    Inside loop: 3  
Starting loop 3:  
    Inside loop: 1
```

Inside loop: 2

Inside loop: 3

\$

Controlling the Loop:

- ▶ The break command
- ▶ The continue command

Each command has a different use in how to control the operation of a loop. The following sections describe how you can use these commands to control the operation of your loops.

- ▶ The break command
- ▶ The break command is a simple way to escape a loop in progress. You can use the break command to exit any type of loop, including while and until loops.
- ▶ You can use the break command in several situations. This section shows each of these methods.

Break command

```
$ cat test17 #!/bin/bash
```

```
# breaking out of a for loop
```

```
for var1 in 1 2 3 4 5 6 7 8 9 10 do
```

```
if [ $var1 -eq 5 ] then
```

```
break
```

```
fi
```

```
echo "Iteration number: $var1" done
```

```
echo "The for loop is completed"
```

Output

```
$ ./test17
```

```
Iteration number: 1
```

```
Iteration number: 2
```

```
Iteration number: 3
```

```
Iteration number: 4
```

```
The for loop is completed
```

```
$
```

The continue command

- ▶ The continue command is a way to prematurely stop processing commands inside of a loop but not terminate the loop completely.
- ▶ This allows you to set conditions within a loop where the shell won't execute commands.
- ▶ Here's a simple example of using the continue command in a for loop:

When the conditions of the if-then statement are met (the value is greater than 5 and less than 10), the shell executes the continue command, which skips the rest of the commands in the loop, but keeps the loop going.

When the if-then condition is no longer met, things return to normal.

Example

```
$ cat test21 #!/bin/bash
```

```
# using the continue command
```

```
for (( var1 = 1; var1 < 15; var1++ ))
```

```
do
```

```
if [ $var1 -gt 5 ] && [ $var1 -lt 10 ] then
```

```
continue
```

fi

echo "Iteration number: \$var1"

done

▶ **\$./test21**

Processing the Output of a Loop :

- ▶ Finally, you can either pipe or redirect the output of a loop within your shell script. You do this by adding the processing command to the end of the done command:

```
for file in /home/rich/* do  
if [ -d "$file" ] then  
echo "$file is a directory" elif  
echo "$file is a file"  
fi  
done > output.txt
```

- ▶ Instead of displaying the results on the monitor, the shell redirects the results of the for command to the file output.txt.
- ▶ Consider the following example of redirecting the output of a for command to a file:

```
$ cat test23 #!/bin/bash  
# redirecting the for output to a file  
for (( a = 1; a < 10; a++ )) do  
echo "The number is $a" done > test23.txt  
echo "The command is finished."
```

▶ **\$./test23**

▶ The command is finished.

▶ **\$ cat test23.txt** The number is 1 The number is 2 The number is 3 The number is 4
The number is 5 The number is 6 The number is 7 The number is 8 The number is
9

- ▶ \$
- ▶ The shell creates the file test23.txt and redirects the output of the for command only to the file. The shell displays the echo statement after the for command just as normal.

2.2 Handling User Input:

- Passing parameters
- Tracking parameters
- Being shifty
- Working with options
- Standardizing options
- Getting user input

Passing Parameters

- The most basic method of passing data to your shell script is **to use *command line parameters***. Command line parameters allow you **to add data values** to the command line when you execute the script:
- \$ **`./addem 10 30`**
- This example passes two command line parameters (10 and 30) to the script **addem**. The script handles the command line parameters using **special variables**. The following sections describe how to use command line parameters in your bash shell scripts.
-

Reading parameters

- The bash shell assigns **special variables, called *positional parameters***, to all of the command line parameters entered.
- This includes the name of the script the shell is executing.
- The positional parameter variables are **standard numbers**, with **\$0 being the script's name, \$1 being the first parameter, \$2 being the second parameter, and so on, up to \$9 for the ninth parameter**.

- A simple example of using one command line parameter in a shell script
- **\$ cat test1.sh**
- `#!/bin/bash`
- `# using one command line parameter #`
- `factorial=1`
- `for ((number = 1; number <= $1 ; number++)) do`
- `factorial=$((factorial * $number) done`
- `echo The factorial of $1 is $factorial`
- `$`
- **\$./test1.sh 5**
- The factorial of 5 is 120
- `$`
- You can use the **\$1 variable just like any other variable** in the shell script.
- The shell script automatically assigns the value from the **command line parameter to the variable**; you don't need to do anything with it.

If you need to enter more command line parameters, each parameter must be **separated by a space** on the command line:

- **\$ cat test2.sh**
- `#!/bin/bash`
- `# testing two command line parameters #`
- `total=$(($1 * $2)`
- `echo The first parameter is $1. echo The second parameter is $2. echo The total value is $total.`
- `$`

Result

- **\$./test2.sh 2 5**

- The first parameter is 2. The second parameter is 5. The total value is 10.
- \$

Reading the script name

You can use the **\$0 parameter to determine the script name** the shell started from the command line. This can come in handy if you're writing a utility that can have multiple functions.

- **\$ cat test5.sh**
- `#!/bin/bash`
- `# Testing the $0 parameter #`
- `echo The zero parameter is set to: $0 #`
- \$
- **\$ bash test5.sh**
- The zero parameter is set to: test5.sh
- \$
- **\$./test5.sh**
- The zero parameter is set to: ./test5.sh
- \$

Testing parameters

Be careful **when using command line parameters** in your shell scripts. If the script is run without the parameters, bad things can happen:

```
$ ./addem 2
```

```
./addem: line 8: 2 +      :
```

```
syntax error: operand expected (error token is " ")
```

```
The calculated value is
```

```
$
```

When the script assumes there is data in a parameter variable, and no data is present, most likely you'll get an error message from your script. This is a poor way to write scripts.

- Always check your parameters to make sure the data is there before using it:
- **\$ cat test7.sh**
- `#!/bin/bash`
- `# testing parameters before use #`
- `if [-n "$1"] then`
- `echo Hello $1, glad to meet you. else`
- `echo "Sorry, you did not identify yourself. "`
- `fi`
- `$`
- **\$./test7.sh Rich**
- `Hello Rich, glad to meet you.`
- `$`
- **\$./test7.sh**
- `Sorry, you did not identify yourself.`
- `$`
- In this example, the `-n` test evaluation was used to check for data in the `$1` command line parameter.

Using_Special_Parameter_Variables

- A few special bash shell variables track command line parameters. This section describes what they are and how to use them.
- **Counting parameters**
- As you saw in the last section, you should verify command line parameters before using them in your script. For scripts that use multiple command line parameters, this checking can get tedious.

- Instead of testing each parameter, you can count how many parameters were entered on the command line. The bash shell provides a special variable for this purpose.

Example

- The special `$#` variable contains the number of command line parameters included when the script was run. You can use this special variable anywhere in the script, just like a normal variable:
- `$ cat test8.sh`
- `#!/bin/bash`
- `# getting the number of parameters #`
- `echo There were $# parameters supplied.`
- `$`

Example

- `$./test8.sh`
- There were 0 parameters supplied.
- `$`
- `$./test8.sh 1 2 3 4 5`
- There were 5 parameters supplied.
- `$`
- `$./test8.sh 1 2 3 4 5 6 7 8 9 10`
- There were 10 parameters supplied.
- `$`
- `$./test8.sh "Rich Blum"`
- There were 1 parameters supplied.
- `$`

Now you have the ability to test the number of parameters present before trying to use them:

- **\$ cat test9.sh**
- `#!/bin/bash`
- `# Testing parameters #`
- `if [$# -ne 2] then`
- `echo`
- `echo Usage: test9.sh a b echo`
- `else`
- `total=$(($1 + $2)) echo`
- `echo The total is $total echo`
- `fi #`

\$

Output

- **\$ bash test9.sh**
- Usage: test9.sh a b
- **\$ bash test9.sh 10**
- Usage: test9.sh a b
- **\$ bash test9.sh 10 15**
- The total is 25
- \$
- The if-then statement uses the `-ne` evaluation to perform a numeric test of the command line parameters supplied. If the correct number of parameters isn't present, an error message displays showing the correct usage of the script.
- **\$ cat test10.sh**
- `#!/bin/bash`
- `# Grabbing the last parameter #`
- `params=${#}`

- echo
- echo The last parameter is \$params
- echo The last parameter is \${!#}
- echo#
- \$
- **\$ bash test10.sh 1 2 3 4 5**
- The last parameter is 5 The last parameter is 5
- \$
- **\$ bash test10.sh**
- The last parameter is 0
- The last parameter is test10.sh
- \$

Grabbing all the data

- In some situations you want to **grab all the parameters** provided on the command line. Instead of having to mess with using the **\$# variable** to determine how many parameters are on the command line and having to loop through all of them, you can **use a couple of other special variables**.
- **The \$* and \$@ variables** provide easy access to all your parameters. Both of these variables include all the command line parameters **within a single variable**.
- The **\$* variable** takes all the parameters supplied on the command line **as a single word**. The word contains each of the values as they appear on the command line. Basically, instead of treating the parameters as **multiple objects**, the **\$*** variable treats them **all as one parameter**.
- The **\$@ variable**, on the other hand, takes **all the parameters supplied on the command line as separate words in the same string**. It allows you to iterate through the values, separating out each parameter supplied. This is most often accomplished using the **for command**.

- The shell sets the **`$#` variable to the number of parameters entered on the command line**. The **`$*` variable contains all the parameters as a single string**, and the **`$@` variable contains all the parameters as separate words**.

Being Shifty

- Another tool you have in your bash shell tool belt is the **shift command**. The bash shell provides the shift command to help you manipulate command line parameters. The **shift command literally shifts the command line parameters in their relative positions**.
- When you use the shift command, **it moves each parameter variable one position to the left by default**. Thus, the value for variable **`$3` is moved to `$2`, the value for variable `$2` is moved to `$1`, and the value for variable `$1` is discarded** (note that the value for variable `$0`, the program name, remains unchanged).

- **`$ cat test13.sh`**
- `#!/bin/bash # demonstrating the shift command echo`
- `count=1`
- `while [-n "$1"]`
- `do`
 - `echo "Parameter #${count} = $1"`
 - `count=$((count + 1)`
 - `shift`

done

- `$`
- **`$./test13.sh rich barbara katie jessica`**
- `Parameter #1 = rich`
- `Parameter #2 = barbara`
- `Parameter #3 = katie`
- `Parameter #4 = jessica`

- \$
- **\$ cat test14.sh**

```
#!/bin/bash

# demonstrating a multi-position shift #

echo

echo "The original parameters: $*" shift 2

echo "Here's the new first parameter: $1"
```
- \$
- **\$./test14.sh 1 2 3 4 5**
 - The original parameters: 1 2 3 4 5 Here's the new first parameter: 3
 - \$
- By using values in the shift command, you can easily skip over parameters you don't need.

Working with Options

- **Options** are single letters preceded by a dash that alter the behavior of a command. This section shows **three methods for working with options** in your shell scripts.
- **Finding your options**
- On the surface, there's nothing all that special about command line options. They appear on the **command line immediately after the script name**, just the **same as command line parameters**.
- In fact, if you want, you can process command line options the **same way you process command line parameters**.

Processing simple options

- In the **test13.sh** script earlier, you saw how to use the **shift command** to work your way down the command line parameters provided with the script program.
- You can use this **same technique to process command line options**.

- **\$ cat test15.sh**
- `#!/bin/bash`
- `# extracting command line options as parameters #`
- `echo`
- `while [-n "$1"] do`
- `case "$1" in`
- `-a) echo "Found the -a option" ;;`
- `-b) echo "Found the -b option" ;;`
- `-c) echo "Found the -c option" ;;`
- `*) echo "$1 is not an option" ;; esac`
- `shift done`
- `$`
- **\$./test15.sh -a -b -c -d**
- Found the -a option Found the -b option Found the -c option -d is not an option
- `$`
- The case statement checks each parameter for valid options. When one is found, the appropriate commands are run in the case statement.
- This method works, no matter in what order the options are presented on the command line:

\$./test15.sh -d -c -a

-d is not an option

Found the -c option

Found the -a option

- Separating options from parametersThe standard way to do this in Linux is to separate the two with a special character code that tells the script when the options are finished and when the normal parameters start.

- For Linux, this special character is the double dash (--). The shell uses the double dash to indicate the end of the option list. After seeing the double dash, your script can safely process the remaining command line parameters as parameters and not options.
- **To check for the double dash, simply add another entry in the case statement:**
 - `$ cat test16.sh`
 - `#!/bin/bash`
 - `# extracting options and parameters echo`
 - `while [-n "$1"] do`
 - `case "$1" in`
 - `-a) echo "Found the -a option" ;;`
 - `-b) echo "Found the -b option";;`
 - `-c) echo "Found the -c option" ;;`
 - `--) shift`
 - `break ;;`
 - `*) echo "$1 is not an option";; esac`
 - `shift done`
 - `#`
 - `count=1`
 - `for param in $@ do`
 - `echo "Parameter #$count: $param" count=$(($count + 1)`
 - `done`
- `$`
- This script uses the break command to break out of the while loop when it encounters the double dash. Because we're breaking out prematurely, we need to

ensure that we stick in another shift command to get the double dash out of the parameter variables.

- **\$./test16.sh -c -a -b test1 test2 test3**
- Found the -c option Found the -a option Found the -b option test1 is not an option
- test2 is not an option test3 is not an option
- **\$**

Processing options with values

- **\$ cat test17.sh**
- **#!/bin/bash**
- **# extracting command line options and values echo**
- **while [-n "\$1"] do**
- **case "\$1" in**
- **-a) echo "Found the -a option";;**
- **-b) param="\$2"**
- **echo "Found the -b option, with parameter value \$param" shift ;;**
- **-c) echo "Found the -c option";;**
- **--) shift**
- **break ;;**
- ***) echo "\$1 is not an option";; esac shift done**
- **#**
- **count=1**
- **for param in "\$@" do**
- **echo "Parameter # \$count: \$param" count=\$((\$count + 1)**
- **done**

- \$
- \$./test17.sh -a -b test1 -d
- Found the -a option
- Found the -b option, with parameter value test1
- -d is not an option
- \$

- In this example, the case statement defines three options that it processes.
- The -b option also requires an additional parameter value. Because the parameter being processed is \$1, you know that the additional parameter value is located in \$2 (because all the parameters are shifted after they are processed).
- Just extract the parameter value from the \$2 variable.
- Of course, because we used two parameter spots for this option, you also need to set the shift command to shift one additional position.

Using the getopt command

- The getopt command is a great tool to have handy when processing command line options and parameters. It reorganizes the command line parameters to make parsing them in your script easier.

Looking at the command format

- The getopt command can take a list of command line options and parameters, in any form, and automatically turn them into the proper format. It uses the following command format:

```
getopt optstring parameters
```

- Here's a simple example of how getopt works:

- `$ getopt ab:cd -a -b test1 -cd test2 test3`
- `-a -b test1 -c -d -- test2 test3`
- `$`
- The *optstring* defines four valid option letters, a, b, c, and d.
- A colon (:) is placed behind the letter b in order to require option b to have a parameter value.
- When the `getopt` command runs, it examines the provided parameter list (`-a -b test1 -cd test2 test3`) and parses it based on the supplied *optstring*.
- Notice that it automatically separated the `-cd` options into two separate options and inserted the double dash to separate the additional parameters on the line.

Using getopt in your scripts

- One of the `set` command options is the double dash (`--`). The double dash instructs `set` to replace the command line parameter variables with the values on the `set` command's command line.
- The trick then is to feed the original script command line parameters to the `getopt` command and then feed the output of the `getopt` command to the `set` command to replace the original command line parameters with the nicely formatted ones from `getopt`.
- This looks something like this:
- `set -- $(getopt -q ab:cd "$@")`
- Now the values of the original command line parameter variables are replaced with the output from the `getopt` command, which formats the command line parameters for us.
- This command converts command line options and parameters into a standard format that you can process in your script.
- The `getopt` command allows you to specify which letters it recognizes as options and which options require an additional parameter value.
- The `getopt` command processes the standard command line parameters and outputs the options and parameters in the proper order.

getopts command

- The final method for handling command line options is via the getopts command (note that it's plural).
- The getopts command provides more advanced processing of the command line parameters.
- It allows for multi-value parameters, along with identifying options not defined by the script.

Standardizing Options

- When you create your shell script, obviously you're in control of what happens. It's completely up to you as to which letter options you select to use and how you select to use them.
- However, a few letter options have achieved a somewhat standard meaning in the Linux world. If you control these options in your shell script, your scripts will be more user-friendly.

Common Linux Command Line Options

| Option | Description |
|--------|--|
| -a | Shows all objects |
| -c | Produces a count |
| -d | Specifies a directory |
| -e | Expands an object |
| -f | Specifies a file to read data from |
| -h | Displays a help message for the command |
| -i | Ignores text case |
| -l | Produces a long format version of the output |

| Option | Description |
|---------------|--|
| -n | Uses a non-interactive (batch) mode |
| -o | Specifies an output file to redirect all output to |
| -q | Runs in quiet mode |
| -r | Processes directories and files recursively |
| -s | Runs in silent mode |
| -v | Produces verbose output |
| -x | Excludes an object |
| -y | Answers yes to all questions |

Getting User Input

- Although providing command line options and parameters is a great way to get data from your script users, sometimes your script needs to be more interactive.
- Sometimes you need to ask a question while the script is running and wait for a response from the person running your script.
- The bash shell provides the read command just for this purpose.
- An interactive method to obtain data from your script users is the read command. The read command allows your scripts to query users for information and wait. The read command places any data entered by the script user into one or more variables, which you can use within the script.
- Several options are available for the read command that allow you to customize the data input into your script, such as using hidden data entry, applying timed data entry, and requesting a specific number of input characters.

Reading basics

- The read command accepts input either from standard input (such as from the keyboard) or from another file descriptor.
- After receiving the input, the read command places the data into a variable.
- Here's the read command at its simplest:

\$ cat test21.sh

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

```
# testing the read command #
```

```
echo -n "Enter your name: " read name
```

```
echo "Hello $name, welcome to my program. "
```

\$

- In fact, the read command includes the -p option, which allows you to specify a prompt directly in the read command line:
- \$ cat test22.sh
- #!/bin/bash
- # testing the read -p option #
- read -p "Please enter your age: " age
- days=\$((\$age * 365))
- echo "That makes you over \$days days old! "

\$./test21.sh

```
Enter your name: Rich Blum
```

```
Hello Rich Blum, welcome to my program.
```

\$

- \$./test22.sh
- Please enter your age: 10
- That makes you over 3650 days old!
- \$
- Timing out
- Be careful when using the read command. Your script may get stuck waiting for the script user to enter data.
- If the script must go on regardless of whether any data was entered, you can use the -t option to specify a timer.

- The -t option specifies the number of seconds for the read command to wait for input.
- \$ cat test25.sh
- #!/bin/bash
- # timing the data entry #
- if read -t 5 -p "Please enter your name: " name
- then
- echo "Hello \$name, welcome to my script"
- else
- echo
- echo "Sorry, too slow! "
- fi
- \$
- \$./test25.sh
- Please enter your name: Rich
- Hello Rich, welcome to my script
- \$
- \$./test25.sh
- Please enter your name:
- Sorry, too slow!
- \$

Reading with no display :

- Sometimes you need input from the script user, but you don't want that input to display on the monitor. The classic example is when entering passwords, but there are plenty of other types of data that you need to hide.
- The -s option prevents the data entered in the read command from being displayed on the monitor; actually, the data is displayed, but the read command sets the text color to the same as the background color. Here's an example of using the -s option in a script:

- `$ cat test27.sh`
- `#!/bin/bash`
- `# hiding input data from the monitor #`
- `read -s -p "Enter your password: " pass echo`
- `echo "Is your password really $pass? "`

example

`$./test27.`

Enter your password:

Is your password really T3st1ng?

`$`

Reading from a file

- Finally, you can also use the read command to read data stored in a file on the Linux system.
- Each call to the read command reads a single line of text from the file.
- When no more lines are left in the file, the read command exits with a non-zero exit status.
- The tricky part is getting the data from the file to the read command.
- The most common method is to pipe the result of the cat command of the file directly to a while command that contains the read command.
- **`$ cat test28.sh`**
 - `#!/bin/bash`
 - `# reading data from a file #`
 - `count=1`
 - `cat test | while read line`
 - `do`

- echo "Line \$count: \$line"
- count=\$((\$count + 1))
- done
- echo "Finished processing the file"
- \$
- \$ cat test
- The quick brown dog jumps over the lazy fox.
- This is a test, this is only a test.
- O Romeo, Romeo! Wherefore art thou Romeo?
- \$
- **\$./test28.sh**
 - Line 1: The quick brown dog jumps over the lazy fox.
 - Line 2: This is a test, this is only a test.
 - Line 3: O Romeo, Romeo! Wherefore art thou Romeo?
 - Finished processing the file
- \$
- The while command loop continues processing lines of the file with the read command, until the read command exits with a non-zero exit status.

Scripting control

Stopping processes

- System administrator is knowing when and how to stop a process.
- Sometimes, a process gets hung up and needs a gentle push to either get going again or stop.
- Other times, a process runs away with the CPU and refuses to give it up.
- In both cases, you need a command that allows you to control a process. Linux follows the Unix method of inter process communication.

- In Linux, processes communicate with each other using *signals*. A *process signal* is a predefined message that processes recognize and may choose to ignore or act on.
- The developers program how a process handles signals. Most well-written applications have the ability to receive and act on the standard Unix process signals.

Linux Process Signals

| Signal | Name | Description |
|--------|------|---|
| 1 | HUP | Hangs up |
| 2 | INT | Interrupts |
| 3 | QUIT | Stops running |
| 9 | KILL | Unconditionally terminates |
| 11 | SEGV | Produces segment violation |
| 15 | TERM | Terminates if possible |
| 17 | STOP | Stops unconditionally, but doesn't terminate |
| 18 | TSTP | Stops or pauses, but continues to run in background |
| 19 | CONT | Resumes execution after STOP or TSTP |

2.3 SCRIPTING CONTROL

- ▶ Linux uses signals to communicate with processes running on the system.
- ▶ You can control the operation of your shell script by programming the script to perform certain commands when it receives specific signals.

Signaling the bash shell

- ▶ There are more than 30 Linux signals that can be generated by the system and applications.
- ▶ Most common Linux system signals that you'll run across in your shell script writing.

| Signal | Value | Description |
|--------|---------|---|
| 1 | SIGHUP | Hangs up the process |
| 2 | SIGINT | Interrupts the process |
| 3 | SIGQUIT | Stop the process |
| 9 | SIGKILL | Unconditionally terminates the process |
| 15 | SIGTERM | Terminate the process if possible |
| 17 | SIGSTOP | Unconditionally stops, but doesn't terminate, the process |
| 18 | SIGTSTP | Stops or pauses the process, but doesn't terminate |
| 19 | SIGCONT | Continue a stopped process |

- ▶ By default, the bash shell ignores any SIGQUIT (3) and SIGTERM (15) signals it receives (so an interactive shell cannot be accidentally terminated).
- ▶ However, the bash shell does not ignore any SIGHUP (1) and SIGINT (2) signals it receives.
- ▶ If the bash shell receives a SIGHUP signal, such as when you leave an interactive shell, it exits.
- ▶ Before it exits, however, it passes the SIGHUP signal to any processes started by the shell, including any running shell scripts.

Generating signals

- ▶ The bash shell allows you to generate two basic Linux signals using key combinations on the keyboard.

- ▶ This feature comes in handy if you need to stop or pause a runaway script.

Interrupting a process:

- ▶ The Ctrl+C key combination generates a SIGINT signal and sends it to any processes currently running in the shell.
- ▶ You can test this by running a command that normally takes a long time to finish and pressing the Ctrl+C key combination:

```
$ sleep 100
```

```
^C
```

```
$
```

Pausing a process

- ▶ Instead of terminating a process, you can pause it in the middle of whatever it's doing.
- ▶ Sometimes, this can be a dangerous thing (for example, if a script has a file lock open on a crucial system file), but often it allows you to peek inside what a script is doing without actually terminating the process.
- ▶ When you use the Ctrl+Z key combination, the shell informs you that the process has been stopped:

```
$ sleep 100
```

```
^Z
```

```
[1]+  Stopped      sleep 100
```

```
$
```

Trapping signals

- ▶ Instead of allowing your script to leave signals ungoverned, you can trap them when they appear and perform other commands.
- ▶ The trap command allows you to specify which Linux signals your shell script can watch for and intercept from the shell.

- ▶ If the script receives a signal listed in the trap command, it prevents it from being processed by the shell and instead handles it locally.
- ▶ The format of the trap command is:

trap *commands signals*

Trapping a script exit

- ▶ Besides trapping signals in your shell script, you can trap them when the shell script exits.
- ▶ This is a convenient way to perform commands just as the shell finishes its job.
- ▶ When the script gets to the normal exit point, the trap is triggered, and the shell executes the command you specify on the trap command line.

▶ **\$ cat test1.sh**

▶ **#!/bin/bash**

```
# Testing signal trapping #

trap "echo ' Sorry! I have trapped Ctrl-C'" SIGINT # echo This is a test
script #

count=1

while [ $count -le 10 ]

do

echo "Loop #$count"

sleep 1

count=$(( $count + 1 )

done

#echo "This is the end of the test script" #

$ ./test1.sh

This is a test script Loop #1

Loop #2

Loop #3
```



```
Loop #4
Loop #5
^C Sorry! I have trapped Ctrl-C Loop #6
Loop #7
Loop #8
^C Sorry! I have trapped Ctrl-C Loop #9
Loop #10
This is the end of the test script
$
```

- ▶ Each time the Ctrl+C key combination was used, the script executed the echo statement specified in the trap command instead of not managing the signal and allowing the shell to stop the script.

Modifying or removing a trap

- ▶ To handle traps differently in various sections of your shell script, you simply reissue the
- ▶ trap command with new options:
- ▶ After the signal trap is modified, the script manages the signal or signals differently. However, if a signal is received before the trap is modified, the script processes it per the original trap command

Running Scripts in Background Mode

- ▶ Sometimes, running a shell script directly from the command line interface is inconvenient.
- ▶ Some scripts can take a long time to process, and you may not want to tie up the command line interface waiting.
- ▶ While the script is running, you can't do anything else in your terminal session.
- ▶ Fortunately, there's a simple solution to that problem.
- ▶ Running in the background

- ▶ Running a shell script in background mode is a fairly easy thing to do. To run a shell
- ▶ script in background mode from the command line interface, just place an ampersand
- ▶ symbol (&) after the command:

```

$ cat test4.sh

#!/bin/bash

# Test running in the background #

count=1

while [ $count -le 10 ]

do

sleep 1

count=$(( $count + 1 )

done

$ ./test4.sh &

[1] 3231

$

```

- ▶ When you place the ampersand symbol after a command, it separates the command from the bash shell and runs it as a separate background process on the system. The first thing that displays is the line:
- ▶ [1] 3231
- ▶ The number in the square brackets is the job number assigned by the shell to the background process. The next number is the Process ID (PID) the Linux system assigns to the process. Every process running on the Linux system must have a unique PID.
- ▶ As soon as the system displays these items, a new command line interface prompt appears. You are returned to the shell, and the command you executed runs safely in background mode. At this point, you can enter new commands at the prompt.
- ▶ When the background process finishes, it displays a message on the terminal:

- ▶ Done ./test4.sh
- ▶ This shows the job number and the status of the job (Done), along with the command used to start the job.

Be aware that while the background process is running, it still uses your terminal monitor for STDOUT and STDERR messages:

```
$ cat test5.sh
```

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

```
# Test running in the background with output #
```

```
echo "Start the test script" count=1
```

```
while [ $count -le 5 ] do
```

```
echo "Loop #$count" sleep 5
```

```
count=$(( $count + 1 )) done
```

```
#
```

```
echo "Test script is complete" #
```

```
$
```

```
$ ./test5.sh &
```

```
[1] 3275
```

```
$ Start the test script Loop #1
```

```
Loop #2
```

```
Loop #3
```

```
Loop #4
```

```
Loop #5
```

```
Test script is complete
```

```
Done ./test5.sh
```

```
$
```

Running Scripts without a Hang-Up

Sometimes, you may want to start a shell script from a terminal session and let the script run in background mode until it finishes, even if you exit the terminal session. You can do this by using the `nohup` command

16

The `nohup` command runs another command blocking any `SIGHUP` signals that are sent to the process. This prevents the process from exiting when you exit your terminal session.

```
$ nohup ./test1.sh &
```

```
[1] 3856
```

```
$ nohup: ignoring input and appending output to 'nohup.out'
```

```
$
```

Controlling the Job

- Earlier in this chapter, you saw how to use the `Ctrl+C` key combination to stop a job running in the shell.
- After you stop a job, the Linux system lets you either kill or restart it.
- You can kill the process by using the `kill` command. Restarting a stopped process requires that you send it a `SIGCONT` signal.
- The function of starting, stopping, killing, and resuming jobs is called *job control*.
- With job control, you have full control over how processes run in your shell environment.
- This section describes the commands used to view and control jobs running in your shell.

Viewing jobs

The key command for job control is the `jobs` command. The `jobs` command allows you to view the current jobs being handled by the shell:

\$ cat test10.sh

```
#!/bin/bash

# Test job control #

echo "Script Process ID: $$" #

count=1

while [ $count -le 10 ] do

echo "Loop #$count" sleep 10

count=$(( $count + 1 ]) done

#

echo "End of script..." #

$
```

Output

\$./test10.sh

Script Process ID: 1897 Loop #1

Loop #2

^Z

+ Stopped ./test10.sh

\$

The jobs Command Parameters

| Parameter | Description |
|------------------|---|
| -l | Lists the PID of the process along with the job number |
| -n | Lists only jobs that have changed their status, since the last notification from the shell |
| -p | Lists only the PIDs of the jobs |

- r Lists only the running jobs
- s Lists only stopped jobs

Restarting stopped jobs

- Under bash job control, you can restart any stopped job as either a background process or a foreground process.
- A foreground process takes over control of the terminal you're working on, so be careful about using that feature.

```

$ ./test11.sh
^Z
+      Stopped      ./test11.sh
$ bg
[1]+ ./test11.sh &
$
$ jobs
[1]+  Running      ./test11.sh &
$

```

- Because the job was the default job, indicated by the plus sign, only the bg command was needed to restart it in background mode. Notice that no PID is listed when the job is moved into background mode.
- If you have additional jobs, you need to use the job number along with the bg command:

```

$ ./test11.sh
^Z
+      Stopped      ./test11.sh
$
$ ./test12.sh
^Z
+      Stopped      ./test12.sh
$
$ bg 2
[2]+ ./test12.sh &
$

```

\$ jobs

```
[1]+  Stopped    ./test11.sh
[2]-  Running    ./test12.sh &
```

▶ \$

- The command `bg 2` was used to send the second job into background mode. Notice that when the `jobs` command was used, it listed both jobs with their status, even though the default job is not currently in background mode.

To restart a job in foreground mode, use the `fg` command, along with the job number:

▶ **\$ fg 2**

▶ `./test12.sh`

▶ This is the script's end...

▶ \$

▶ Because the job is running in foreground mode, the command line interface prompt does not appear until the job finishes.

Being Nice

- In a multitasking operating system (which Linux is), the kernel is responsible for assigning CPU time for each process running on the system.
- The ***scheduling priority*** is the amount of CPU time the kernel assigns to the process relative to the other processes.
- By default, all processes started from the shell have the same scheduling priority on the Linux system.
- Sometimes, you want to change the priority of a shell script, either lowering its priority so it doesn't take as much processing power away from other processes or giving it a higher priority so it gets more processing time.
- You can do this by using the **nice command**.

Using the nice command

- The `nice` command allows you to set the scheduling priority of a command as you start it.

- To make a command run with less priority, just use the `-n` command line option for `nice` to specify a new priority level:

```
$ nice -n 10 ./test4.sh > test4.out &
```

```
[1] 4973
```

```
$
```

```
$ ps -p 4973 -o pid,ppid,ni,cmd
```

```
PID  PPID  NI CMD
```

```
4973  4721  10 /bin/bash ./test4.sh
```

```
▶ $
```

- ▶ Sometimes, you'd like to change the priority of a command that's already running on the system. That's what the `renice` command is for. It allows you to specify the PID of a running process to change its priority:

- ▶ `$./test11.sh &`

```
[1] 5055
```

```
$
```

```
$ ps -p 5055 -o pid,ppid,ni,cmd
```

```
PID  PPID  NI CMD
```

```
5055  4721  0 /bin/bash ./test11.sh
```

```
$
```

```
$ renice -n 10 -p 5055
```

```
5055: old priority 0, new priority 10
```

```
$
```

```
$ ps -p 5055 -o pid,ppid,ni,cmd
```

```
PID  PPID  NI CMD
```

```
5055  4721  10 /bin/bash ./test11.sh
```

```
$
```

- The `renice` command automatically updates the scheduling priority of the running process.
- As with the `nice` command, the `renice` command has some limitations:
- You can only `renice` processes that you own.
- You can only `renice` your processes to a lower priority.
- The root user can `renice` any process to any priority.

- If you want to fully control running processes, you must be logged in as the root account or use the sudo command.
- Sudo (superuser do) is a utility for UNIX- and Linux-based systems that provides an efficient way to give specific users permission to use specific system commands at the root (most powerful) level of the system. Sudo also logs all commands and arguments.

Scheduling a job using the at command

- The at command allows you to specify a time when the Linux system will run a script.
- The at command submits a job to a queue with directions on when the shell should run the job. The at daemon, atd, runs in the background and checks the job queue for jobs to run.
- Most Linux distributions start this daemon automatically at boot time.
- Understanding the at command format
- The basic at command format is pretty simple:
- `at [-f filename] time`
- By default, the at command submits input from STDIN to the queue. You can specify a file- name used to read commands (your script file) using the -f parameter.
- The *time* parameter specifies when you want the Linux system to run the job. If you specify a time that has already passed, the at command runs the job at that time on the next day.
- You can get pretty creative with how you specify the time. The at command recognizes lots of different time formats:
- A standard hour and minute, such as 10:15
- An AM/PM indicator, such as 10:15PM
- A specific named time, such as now, noon, midnight, or teatime (4PM)
- Retrieving job output
- When the job runs on the Linux system, there's no monitor associated with the job.
- Instead, the Linux system uses the e-mail address of the user who submitted the job as STDOUT and STDERR.
- Any output destined to STDOUT or STDERR is mailed to the user via the mail system.

Scheduling regular scripts

- ▶ Using the `at` command to schedule a script to run at a preset time is great, but what if you need that script to run at the same time every day or once a week or once a month?
- ▶ Instead of having to continually submit `at` jobs, you can use another feature of the Linux system.

Starting scripts with a new shell

- The ability to run a script every time a user starts a new bash shell (even just when a specific user starts a bash shell) can come in handy. Sometimes, you want to set shell features for a shell session or just ensure that a specific file has been set.
- Recall the startup files run when a user logs into the bash shell (covered) Also, remember that not every distribution has all the startup files.
- Essentially, the first file found in the following ordered list is run and the rest are ignored:

`$HOME/.bash_profile`

`$HOME/.bash_login`

`$HOME/.profile`

Unit Summary

The Linux system allows you to control your shell scripts by using signals. The bash shell accepts signals and passes them on to any process running under the shell process. Linux signals allow you to easily kill a runaway process or temporarily pause a long-running process.

You can use the `trap` statement in your scripts to catch signals and perform commands. This feature provides a simple way to control whether a user can interrupt your script while it's running.

By default, when you run a script in a terminal session shell, the interactive shell is suspended until the script completes. You can cause a script or command to run in background mode by adding an ampersand sign (`&`) after the command name. When you run a script

or command in background mode, the interactive shell returns, allowing you to continue entering more commands. Any background processes run using this method are still tied to the terminal session. If you exit the terminal session, the background processes also exit.

To prevent this from happening, use the `nohup` command. This command intercepts any signals intended for the command that would stop it — for example, when you exit the terminal session. This allows scripts to continue running in background mode even if you exit the terminal session. When you move a process to background mode, you can still control what happens to it. The `jobs` command allows you to view processes started from the shell session. After you know the job ID of a background process, you can use the `kill` command to send Linux signals to the process or use the `fg` command to bring the process back to the foreground in the shell session. You can suspend a running foreground process by using the `Ctrl+Z` key combination and place it back in background mode, using the `bg` command.

Let us sum up

Structured commands refer to a set of commands that are organized in a systematic and logical manner. These commands are often used in command-line interfaces (CLIs) and scripts to perform tasks or automate processes.

Script control involves writing and managing scripts to automate tasks and control system behavior. Scripts can be written in various scripting languages, such as Bash, PowerShell, Python, and others.

Check your progress

1. Which syntax is correct for a basic for loop in bash?*

- A) for i in list; do ... done B) for (i=0; i<list; i++); do ... done
C) for i: list; { ... } D) foreach i (list); do ... done

2. What is the key difference between until and while loops in bash?*

- A) until loops execute until a condition is true, while loops execute while a condition is true.

B) until loops are used for infinite loops, while loops are not.

C) until loops do not support conditions, while loops do.

D) until loops can only be used with integers, while loops can be used with any data type.

3. Which of the following is the correct syntax for a while loop in bash?*

A) while (condition); do ... done

B) while [condition]; do ... done

C) while condition; do ... done

D) while { condition }; do ... done

4. Which command can be used to exit from the inner loop of nested loops in bash?*

A) exit

B) break

C) continue

D) return

5. How do you redirect the output of a loop to a file in bash?*

A) for i in list; do ... done > output.txt

B) for i in list; do > output.txt ... done

C) for i in list; > output.txt do ... done

D) for i in list; do ... > output.txt done

6. How do you access the first parameter passed to a bash script?*

A) \$0

B) \$1

C) \$2

D) \$#

7. Which variable holds the total number of parameters passed to a bash script?*

A) \$#

B) \$0

C) \$*

D) \$@

8. What does the shift command do in a bash script?*

A) Shifts the command-line arguments to the left.

B) Shifts the command-line arguments to the right.

C) Clears all command-line arguments.

D) Resets the script's parameters.

9. Which command is typically used to parse command-line options in a bash script?*
- A) getopt B) getopt C) parseopts D) optparse
10. What is the typical format for specifying options in a bash script?*
- A) -option B) /option C) --option D) option=
11. Which command is used to get user input in a bash script?*
- A) read B) input C) get D) scan
12. Which command is used to trap signals in a bash script?*
- A) trap B) catch C) signal D) handle
13. Which symbol is used to run a script in the background?*
- A) & B) && C) | D) ||
14. Which command prevents a script from being terminated by a hang-up signal?*
- A) nohup B) nohangup C) nohup -i D) prevent-hangup
15. Which command is used to bring a background job to the foreground?*
- A) fg B) bg C) jobs D) kill
16. Which command is used to start a script with a lower priority?*
- A) low B) renice C) nice D) setpriority
17. Which command is used to schedule scripts to run at specific times?*
- A) schedule B) at C) crontab D) timer

Here are the answers:

1. A) 2. A) 3. B) 4. B) 5. A) 6. B) 7. A) 8. A) 9. A) 10. C)
11. A) 12. A) 13. A) 14. A) 15. A) 16. C) 17. C)

Self Assessment Questions :

1. Compare C-Style command and multiple test command.
2. Handling user input in linux example.
3. How to use job command in linux.
4. Differentiate between break and continue with example program.
5. Explain about looping statements with examples.
6. How to get user input with multiple options.
7. Explain script control commands with example.
8. How to handling user input? Explain in detail.

Open source e-content links

<https://www.slideshare.net/slideshow/linux-commands-and-file-structure/72805453>

<https://linuxconfig.org/handling-user-input-in-bash-scripts>

<https://youtu.be/42iQKuQodW4?si=ioDEWcbiRvFQ-HfY>

Glossary

ls: Lists the contents of a directory.

Usage: ls [options] [directory]

Options: -l (long format), -a (include hidden files), -h (human-readable sizes).

cd: Changes the current directory.

Usage: cd [directory]

pwd: Prints the current working directory.

Usage: pwd

mkdir: Creates a new directory.

Usage: mkdir [directory]

rmdir: Removes an empty directory.

Usage: rmdir [directory]

rm: Removes files or directories.

Usage: rm [options] [file/directory]

Options: -r (recursive), -f (force).

cp: Copies files or directories.

Usage: cp [options] [source] [destination]

Options: -r (recursive), -i (interactive).

mv: Moves or renames files or directories.

Usage: mv [source] [destination]

touch: Creates an empty file or updates the timestamp of an existing file.

Usage: touch [file]

tar: Archives files.

Usage: tar [options] [archive] [files]

Options: -c (create), -x (extract), -t (list), -z (gzip compression).

echo: Displays a line of text or the value of a variable.

Usage: echo [text]

read: Reads a line of input from the user.

Books

1. Learning the bash Shell: Unix Shell Programming by Cameron Newham and Bill Rosenblatt

2. Bash Cookbook: Solutions and Examples for Bash Users" by Carl Albing, JP Vossen, and Cameron Newham

UNIT – III

Objectives:

- To write modular and reusable code by defining functions that encapsulate specific tasks or operations.
- To create scripts that interact with or automate tasks in graphical desktop environments, enhancing user productivity and system management.
- To use sed and awk for powerful text processing, allowing for advanced manipulation and extraction of data from text files and streams.

3.1 Creating Functions

A function is a collection of statements that execute a specified task. Its main goal is to break down a complicated procedure into simpler subroutines that can subsequently be used to accomplish the more complex routine. For the following reasons, functions are popular.

- Assist with code reuse.
- Enhance the program's readability.
- Modularize the software.
- Allow for easy maintenance.

Types of Functions

The functions in shell scripting can be boxed into a number of categories. The following are some of them:

1. The functions that return a value to the caller. The return keyword is used by the functions for this purpose.

- ✚ Function used to calculate the average of the given numbers.

Example

```
find_avg(){
len=$#
sum=0
for x in "$@"
```

```

do
sum=$((sum + x))
done
avg=$((sum/len))
return $avg
}
find_avg 30 40 50 60
printf "%f" "$?"
printf "\n"

```

Output

```

(base) aayussss2101@aayussss2101-VivoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$ bash file.sh
45.000000
(base) aayussss2101@aayussss2101-VivoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$ █

```

 The functions that terminate the shell using the exit keyword.

Example

```

is_odd(){
x=$1
if [ $((x%2)) == 0 ]; then
echo "Invalid Input"
exit 1
else
echo "Number is Odd"
fi
}
is_odd 64

```

Output

```

(base) aayussss2101@aayussss2101-VivoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$ bash file.sh
Invalid Input
(base) aayussss2101@aayussss2101-VivoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$ █

```

 The functions that alter the value of a variable or variables.

Example

```
a=1
increment(){
a=$((a+1))
return
}
increment
echo "$a"
```

Output

```
(base) aayussss2101@aayussss2101-VlvoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$ bash file.sh
2
(base) aayussss2101@aayussss2101-VlvoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$
```

- ✚ The functions that echo output to the standard output.

Example

```
hello_world(){
echo "Hello World"
return
}
hello_world
```

Output

```
(base) aayussss2101@aayussss2101-VlvoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$ bash file.sh
Hello World
(base) aayussss2101@aayussss2101-VlvoBook-ASUSLaptop-X512FL-X512FL:~/Desktop/geeksforgeeks$
```

Returning a value

Example: Bash function returns multiple values, passed via command substitution,
with a READ command and Here string in calling script

```
#!/usr/bin/env bash
set -e #Stop on any errors
function inside_function() {
yvariable1="8"
yvariable2="64"
echo $yvariable1 $yvariable2
}
read -r ans1 ans2 <<< $(inside_function)
echo "ans1 = $ans1"
echo "ans2 = $ans2"
```

OUTPUT

```
ans1 = 8
ans2 = 64
```

Using variables in functions

A shell variable is a character string in a shell that stores some value. It could be an integer, filename, string, or some shell command itself.

Rules for variable definition

A variable name could contain any alphabet (a-z, A-Z), any digits (0-9), and an underscore (_).

- **Valid Variable Names**

```
ABC
_AV_3
AV232
```

- **Invalid variable names**

```
2_AN
!ABD
$ABC
&QAID
```

Variable Types

There are three main types of variables:

1) Local Variable

Variables which are specific to the current instance of shell. They are basically used within the shell, but not available for the program or other shells that are started from within the current shell.

For example

```
`name=Jayesh`
```

In this case the local variable is (name) with the value of Jayesh. Local variables is temporary storage of data within a shell script.

2) Environment Variable

These variables are commonly used to configure the behavior script and programs that are run by shell. Environment variables are only created once, after which they can be used by any user.

For example

```
`export PATH=/usr/local/bin:$PATH` would add `/usr/local/bin` to the beginning of the shell's search path for executable programs.
```

3) Shell Variables

Variables that are set by shell itself and help shell to work with functions correctly. It contains both, which means it has both, some variables are Environment variable, and some are Local Variables.

For example

```
`$PWD` = Stores working directory
```

```
`$HOME` = Stores user's home directory
```

```
`$SHELL` = Stores the path to the shell program that is being used.
```

Array and variable functions

Each value in an array is indexed starting from index 0 for the first value. It is almost similar to Shell Script but with a slight difference. The difference between an array and a Shell Script array is that it supports values of all data types.

Two types of arrays in the shell script.

❖ **Associative Arrays:** It contains Elements with key-value pairs.

❖ **Indexed Arrays:** It contains Indexed Elements starting with zero.

+ **Variable Assignment:** To assign a value to a variable, you can use the equals sign (=).

```
my_variable="Hello, World!"
```

+ **Variable Access:** To access the value of a variable, you prefix it with a dollar sign (\$).

```
echo $my_variable
```

+ **Array Declaration:** We can declare an array by using parentheses and storing values inside it.

```
my_array=("apple" "banana" "cherry")
```

+ **Access Array Elements:** To access individual elements of an array, use the index inside square brackets.

```
echo ${my_array[0]} # Prints "apple"
```

+ **Array Length:** To get the number of elements in an array, use the # symbol.

```
length=${#my_array[@]}  
echo "Array length: $length"
```

+ **Loop through an Array:** We can use a for loop to iterate through the elements of an array.

```
for item in "${my_array[@]}; do  
echo $item  
done
```

+ **Adding Elements to an Array:** We can add elements to an array using the += operator.

```
my_array+=("grape")
```

+ **Removing Elements from an Array:** Use the unset command to remove an element from an array.

```
unset my_array[1] # Removes the second element (banana)
```

+ **Check if a Variable is Set:** Use the -v flag with if statements to check if a variable is set.

```
if [ -v my_variable ]; then  
echo "my_variable is set"
```

```
fi
```

+ **Conditional Statements:** We can use conditional statements like if, elif, and else to make decisions based on variable values.

```
if [ $my_variable == "Hello" ]; then
    echo "The variable is 'Hello'"
elif [ $my_variable == "World" ]; then
    echo "The variable is 'World'"
else
    echo "The variable is something else"
fi
```

Recursive functions

A recursive function is a function that calls itself from inside itself. This function is very useful when you need to call the function to do something again from inside of it.

To calculate the factorial of 4, you multiply the number by the descending numbers. You can do it like this: The ! sign means factorial.

$4! = 4*3*2*1$

Example

```
#!/bash/bin
factorial()
{
    let n=$1
    if (( "$n" <= "1" ))
    then return 1
    else
        factorial n-1
    return $n*$?
    fi
    return 0
}
factorial 5
echo "factorial 5 = $?"
```

Creating a library

Shell Function Library is basically a collection of functions that can be accessed from anywhere in the development environment. It actually makes shell scripting a bit less tedious and repetitive. By creating a shell script with some functions defined in it, we can then access and call those functions from other files or scripts. It helps in avoiding repeating the code in large files and complex scripts.

Creating function library:

```
#!/bin/bash
function square(){
    v1=$1
    n=$(( $v1 * $v1 ))
    echo $n
}
function expo(){
    v1=$1
    v2=$2
    n=$(( $v1 ** $v2 ))
    echo $n
}
function factorial(){
    v1=$1
    n=1
    while [[ $v1 -gt 0 ]]; do
        n=$(( $n * $v1 ))
        v1=$(( $v1 - 1 ))
    done
    echo $n
}
```

Using Functions From Library:

We need a place or file where we can use or utilize this function library. So we create a shell script to call these functions and use it to avoid repetitive tasks and code.

Example

```
#!/bin/bash
echo "4^6 = "$(expo 4 6)
a=5
echo "$a! = "$(factorial $a)
b=18
echo "$b^2 = "$(square $b)
```

Output

```
kumar_satyam@satyam:~/gfg$ source help.sh
4^6 = 4096
5! = 120
18^2 = 324
```

Using functions on the command line

- ✚ **Defining a Function:** We can define a function using the function keyword or simply by using parentheses.

Here's an example using both methods

```
# Using the "function" keyword
my_function() {
    echo "Hello from my_function"
}

# Using parentheses
another_function() {
    echo "Hello from another_function"
}
```

- ✚ **Calling a Function:** We can call a function by its name, followed by parentheses.

Example

```
my_function
another_function
```

- ✚ **Passing Arguments to Functions:** We can pass arguments to functions by referencing them with \$1, \$2, and so on, inside the function.

Example

```
greet() {  
    echo "Hello, $1!"  
}
```

greet "Alice" # Call the function with an argument

✚ **Returning Values from Functions:** We can use the return statement to return a value from a function.

✚ them with \$1, \$2, and so on, inside the function.

Example

```
add() {  
    result=$(( $1 + $2 ))  
    return $result  
}
```

add 3 4 # Call the function to add two numbers

sum=\$? # Get the return value

echo "Sum is \$sum"

Using Functions on the Command Line:

We can use functions on the command line by defining them in your shell script and then sourcing the script to make the functions available in your current shell session.

For example, if you have a script named my_functions.sh with the functions.

```
#!/bin/bash  
greet() {  
    echo "Hello, $1!"  
}
```

3.2 Writing Scripts for Graphical Desktops

1. Desktop Environment:

- Different desktop environments may have their own scripting languages or tools. For example:
 - **GNOME:** You can use GNOME Shell extensions with JavaScript or use the **gsettings** command to configure settings.
 - **KDE Plasma:** Plasma desktop uses QtScript, Python, and D-Bus for scripting.
 - **Xfce:** Xfce supports scripting with Xfce4-panel plugins, including Python.

- **Unity:** You can use Unity's API with Python or other scripting languages.

2. Bash Script with GUI Libraries:

- We can use Bash scripts in conjunction with GUI libraries like Zenity or Yade to create simple GUI dialogs or windows. These libraries allow you to display information, get user input, and provide a basic GUI for your script.

3. Keyboard Shortcuts and Commands:

- We can create shell scripts to automate tasks by assigning keyboard shortcuts to execute them. This can be done through the desktop environment's keyboard shortcut settings.

4. Automating GUI Applications:

- We can use tools like **xdotool** to automate interactions with GUI applications. For example, we can use **xdotool** to simulate mouse clicks, keyboard input, and window management.

5. Desktop Configuration:

- We can use shell scripts to manage desktop configurations, change wallpapers, set themes, and modify desktop-specific settings.

6. Creating Custom Desktop Widgets or Applets:

- Some desktop environments allow you to create custom widgets or applets that can be developed using specific scripting languages or libraries.

Creating text menus

The `select` command to create a simple menu in the terminal. Then, the command displays a list of options preceded by numbers. `Select repeatedly` reads a number from standard input. Subsequently, if the number corresponds to a string's position in `WORDS`, the command sets `NAME` to the respective text.

Steps included:

1. Create a custom menu using **echo** statement and show the menu
2. Create an infinite loop using **while** statement that accept the user input option and generate the output continuously until the user input matches the exit pattern.

3. Take input from the user using **read** statement and store it in a variable.
4. Use case statement to check if the input matches with the pattern.
5. Create custom pattern.
6. Exit the case statement using **esac** keyword.

Example

```
#!/bin/bash
# creating a menu with the following options
echo "SELECT YOUR FAVORITE FRUIT";
echo "1. Apple"
echo "2. Grapes"
echo "3. Mango"
echo "4. Exit from menu "
echo -n "Enter your menu choice [1-4]: "
# Running a forever loop using while statement
# This loop will run until select the exit option.
# User will be asked to select option again and again
while :
do
# reading choice
read choice
# case statement is used to compare one value with the multiple cases.
case $choice in
# Pattern 1
1) echo "You have selected the option 1"
    echo "Selected Fruit is Apple. ";;
# Pattern 2
2) echo "You have selected the option 2"
    echo "Selected Fruit is Grapes. ";;
# Pattern 3
3) echo "You have selected the option 3"
    echo "Selected Fruit is Mango. ";;
# Pattern 4
4) echo "Quitting ..."
    exit;;
```

```

# Default Pattern
*) echo "invalid option";;
esac
echo -n "Enter your menu choice [1-4]: "
done

```

Output

```

SELECT YOUR FAVORITE FRUIT
1. Apple
2. Grapes
3. Mango
4. Exit from menu
Enter your menu choice [1-4]: 1
You have selected the option 1
Selected Fruit is Apple.
Enter your menu choice [1-4]: 2
You have selected the option 2
Selected Fruit is Grapes.
Enter your menu choice [1-4]: 3
You have selected the option 3
Selected Fruit is Mango.
Enter your menu choice [1-4]: 4
Quiting ...

```

Building text window widgets

Install 'dialog' if it's not already installed

On Debian/Ubuntu

```
sudo apt-get install dialog
```

On Red Hat-based systems:

```
sudo yum install dialog
```

Example of a shell script that creates a text window widget using 'dialog'

```

#!/bin/bash
# Function to display a text window
show_text_window() {
    local text_file="$1"
    dialog --textbox "$text_file" 20 60
}
# Main script

```

```
text_file="example.txt"
echo "This is the content of the text window." > "$text_file"
show_text_window "$text_file"
# Clean up the temporary text file
rm -f "$text_file"
```

Adding X Window graphics

Install X Window System: Most Linux distributions come with X11 pre-installed. However, if it's not installed, you can typically install it using your distribution's package manager. For example, on Debian-based systems like Ubuntu,

Code: `sudo apt-get install xorg`

Start X Server: Once X11 is installed, you can start the X server by running the following command.

Code: `startx`

This command will start the X server and bring up a minimal graphical environment with a terminal window.

Run Graphical Applications: You can run graphical applications from the terminal or a launcher. For example, you can run the file manager, text editor, or any other GUI application. Simply type the name of the application and press Enter. For instance:

Code:

`nautilus` # To open the file manager (GNOME)

`gedit` # To open the text editor (GNOME)

Customize the Desktop Environment: Most Linux distributions offer various desktop environments like GNOME, KDE, Xfce, etc. You can choose a desktop environment that suits your preferences and install it. For example, to install the GNOME desktop environment, you can use:

Code:

`sudo apt-get install gnome`

Window Management: The X Window System provides various window management functions, such as moving, resizing, and closing windows. You can also

switch between different virtual desktops, manage workspaces, and more. The exact steps and keybindings for window management depend on the desktop environment you're using.

X Display Server Configuration: You can configure the X server using the Xorg configuration files. These files are usually located in the `/etc/X11/` directory. You may need to customize these files to set display resolutions, input devices, or other hardware-specific settings.

3.3 Introducing sed and gawk

Both 'awk' and 'sed' rely heavily on regular expressions to describe patterns in text upon which some operation should be performed.

Common 'sed' Concepts and Examples

'sed' (Stream Editor)

'sed' is a stream editor that allows you to perform basic text transformations on an input stream (a file or input from a pipeline). It's often used for tasks like search and replace, text substitution, and basic text manipulation.

Search and Replace: You can use 's' (substitute) command to search for a pattern and replace it with another.

Code

```
# Replace "old" with "new" in a file.txt
sed 's/old/new/' file.txt
```

Delete Lines: You can use 'd' command to delete lines that match a pattern.

Code

```
# Delete lines containing "pattern" from file.txt
sed '/pattern/d' file.txt
```

Print Specific Lines You can use line addresses to specify which lines to apply a command to.

Code

```
# Print lines 5 to 10 from file.txt
sed -n '5,10p' file.txt
```

Multiple Commands You can apply multiple 'sed' commands together by separating them with semicolons.

Code

```
# Replace "old" with "new" and delete lines containing "pattern"  
sed -e 's/old/new/' -e '/pattern/d' file.txt
```

'gawk' (GNU Awk): 'gawk' is an enhanced version of the classic 'awk' utility. It's a text processing tool that allows you to process structured text data, typically in the form of records and fields. 'gawk' is particularly powerful for data extraction, manipulation, and reporting.

Common 'gawk' Concepts and Examples

Basic 'gawk' Usage: To use 'gawk' you typically specify a pattern-action pair.

Code

```
# Print the first field of each line  
gawk '{ print $1 }' file.txt
```

Field Separators: By default, 'gawk' uses space as the field separator. You can change the field separator using the '-F' option

Code

```
# Use ":" as the field separator  
gawk -F ':' '{ print $1, $3 }' file.txt
```

Built-in Variables: 'gawk' provides many built-in variables, like 'NF' (number of fields) and 'NR' (record number).

Code

```
# Print the line number and the number of fields  
gawk '{ print NR, NF }' file.txt
```

Regular Expressions: We can use regular expressions in 'gawk' for pattern matching and more complex text processing tasks.

Code

```
# Print lines containing "pattern" anywhere in the text  
gawk '/pattern/ { print }' file.txt
```


Custom Functions: We can define custom functions in 'gawk' to perform complex operations on data.

Code

```
# Define a function and use it to process data
gawk '
function square(x) {
    return x * x
}
{ print square($1) }' file.txt
```

Learning about the sed Editor

SED command in UNIX stands for stream editor and it can perform lots of functions on file like searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace. By using SED you can edit files even without opening them, which is much quicker way to find and replace something in file, than first opening that file in VI Editor and then changing it.

- SED is a powerful text stream editor. Can do insertion, deletion, search and replace(substitution).
- SED command in unix supports regular expression which allows it perform complex pattern matching.

Syntax:

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

- ✚ **Replacing or substituting string:** Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word "unix" with "linux" in the file.

Syntax:

```
$sed 's/unix/linux/' geekfile.txt
```

Output

```
linux is great os. unix is opensource. unix is free os.
```

```
learn operating system.
```

```
linux linux which one you choose.
```

```
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

- Here the “s” specifies the substitution operation. The “/” are delimiters. The “unix” is the search pattern and the “linux” is the replacement string.
- By default, the sed command replaces the first occurrence of the pattern in each line and it won’t replace the second, third...occurrence in the line.

+ Replacing the nth occurrence of a pattern in a line : Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word “unix” with “linux” in a line.

Syntax:

```
$sed 's/unix/linux/2' geekfile.txt
```

Output

```
unix is great os. linux is opensource. unix is free os.
learn operating system.
unix linux which one you choose.
unix is easy to learn.linux is a multiuser os.Learn unix .unix is a
powerful.
```

+ Replacing all the occurrence of the pattern in a line: The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

Syntax:

```
$sed 's/unix/linux/g' windowfile.txt
```

Output

```
linux is great os. linux is opensource. linux is free os.
learn operating system.
linux linux which one you choose.
linux is easy to learn.linux is a multiuser os.Learn linux .linux is a
powerful.
```

+ Replacing from nth occurrence to all occurrences in a line: Use the combination of /1, /2 etc and /g to replace all the patterns from the nth occurrence of a pattern in a line. The following sed command replaces the third, fourth, fifth... “unix” word with “linux” word in a line.

Syntax:

```
$sed 's/unix/linux/3g' windowfile.txt
```

Output

```
unix is great os. unix is opensource. linux is free os.
```

learn operating system.

linux linux which one you choose.

linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.

- + **Parenthesize first character of each word:** This sed example prints the first character of every word in parenthesis.

Syntax:

```
$ echo "Welcome To The Geek Stuff" | sed 's^\(b[A-Z]\)\^(1)/g'
```

Output

(W)elcome (T)o (T)he (G)eek (S)tuff

- + **Replacing string on a specific line number:** You can restrict the sed command to replace the string on a specific line number.

Syntax:

```
$sed '3 s/unix/linux/' geekfile.txt
```

Output

linux is great os. linux is opensource. linux is free os.

learn operating system.

linux linux which one you choose.

linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.

➤The above sed command replaces the string only on the third line.

- + **Duplicating the replaced line with /p flag:** The /p print flag prints the replaced line twice on the terminal. If a line does not have the search pattern and is not replaced, then the /p prints that line only once.

Syntax:

```
$sed 's/unix/linux/p' geekfile.txt
```

Output

linux is great os. linux is opensource. linux is free os.

linux is great os. linux is opensource. linux is free os.

learn operating system.

linux linux which one you choose.

linux linux which one you choose.

linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.

linux is easy to learn.linux is a multiuser os.Learn linux .linux is a powerful.

- + **Printing only the replaced lines:** Use the -n option along with the /p print flag to display only the replaced lines. Here the -n option suppresses the duplicate rows generated by the /p flag and prints the replaced lines only one time.

Syntax:

```
$sed -n 's/unix/linux/p' geekfile.txt
```

Output

```
linux is great os. unix is opensource. unix is free os.
```

```
linux linux which one you choose.
```

```
linux is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

- If you use -n alone without /p, then the sed does not print anything.

- + **Replacing string on a range of lines:** You can specify a range of line numbers to the sed command for replacing a string.

Syntax:

```
$sed '1,3 s/unix/linux/' geekfile.txt
```

Output

```
linux is great os. unix is opensource. unix is free os.
```

```
learn operating system.
```

```
linux linux which one you choose.
```

```
unix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
```

- Here the sed command replaces the lines with range from 1 to 3.

- + **Deleting lines from a particular file:** SED command can also be used for deleting lines from a particular file. SED command is used for performing deletion operation without even opening the file

Examples

To delete a particular line say n in this example

Syntax: \$ sed 'nd' filename.txt

Example:

```
$ sed '5d' filename.txt
```

To Delete a last line

Syntax:

```
$ sed '$d' filename.txt
```

To Delete line from range x to y

Syntax: \$ sed 'x,yd' filename.txt

Example:

```
$ sed '3,6d' filename.txt
```

To Delete from nth to last line

Syntax: \$ sed 'nth,\$d' filename.txt

Example:

```
$ sed '12,$d' filename.txt
```

To Delete pattern matching line

Syntax: \$ sed '/pattern/d' filename.txt

Example:

```
$ sed '/abc/d' filename.txt
```

Getting introduced to the gawk

gawk command in Linux is used for pattern scanning and processing language. The awk command requires no compiling and allows the user to use variables, numeric functions, string functions, and logical operators. It is a utility that enables programmers to write tiny and effective programs in the form of statements that define text patterns that are to be searched for, in a text document and the action that is to be taken when a match is found within a line.

gawk command can be used to :

Scans a file line by line.

- Splits each input line into fields.
- Compares input line/fields to pattern.
- Performs action(s) on matched lines.
- Transform data files.
- Produce formatted reports.
- Format output lines.
- Arithmetic and string operations.
- Conditionals and loops.

Syntax:

```
gawk [POSIX / GNU style options] -f progfile [--] file ...
```

```
gawk [POSIX / GNU style options] [--] 'program' file ...
```

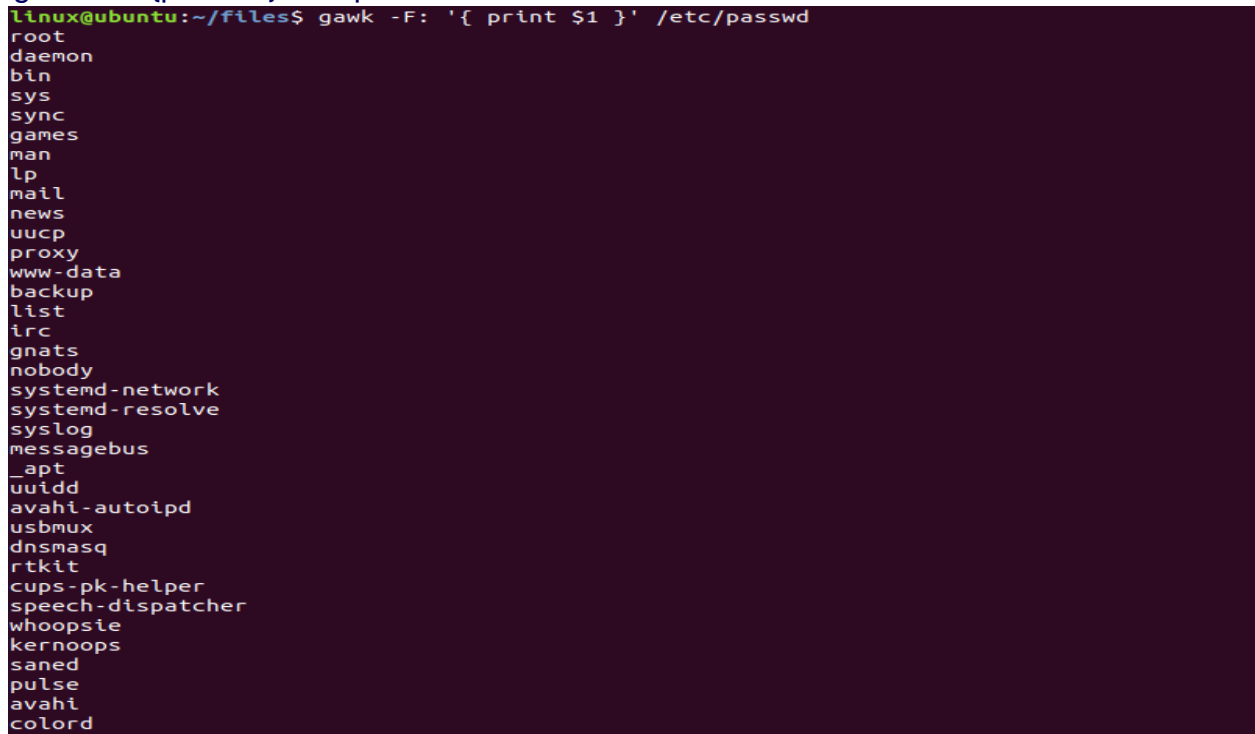
Important Options:

- **-f progfile, --file=progfile:** Read the AWK program source from the file program-file, instead of from the first command line argument. Multiple -f (or --file) options may be used.
- **-F fs, --field-separator=fs:** It uses FS for the input field separator (the value of the FS predefined variable).
- **-v var=val, --assign=var=val:** Assign the value *val* to the variable *var*, before execution of the program begins.

Examples:

- **-F:** It uses FS for the input field separator (the value of the FS predefined variable).

```
gawk -F: '{print $1}' /etc/passwd
```



```
linux@ubuntu:~/files$ gawk -F: '{ print $1 }' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
news
uucp
proxy
www-data
backup
list
irc
gnats
nobody
systemd-network
systemd-resolve
syslog
messagebus
_apt
uuid
avahi-autoipd
usbmux
dnsmasq
rtkit
cups-pk-helper
speech-dispatcher
whoopsie
kernoops
saned
pulse
avahi
colord
```

- **-f:** Read the AWK program source from the file program-file, instead of from the first command line argument. Multiple -f (or --file) options may be used.

```
gawk -F: -f mobile.txt /etc/passwd
```

```

linux@ubuntu:~/files$ gawk -F: -f mobile.txt /etc/passwd
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin

```

Built In Variables:

- **NR:** It keeps a current count of the number of input line.
- **NF:** It keeps a count of the number of fields within the current input record.
- **FS:** It contains the field separator character which is used to divide fields on the input line.
- **RS:** It stores the current record separator character.
- **OFS:** It stores the output field separator, which separates the fields when Awk prints them.
- **ORS:** It stores the output record separator, which separates the output lines when Awk prints them.

Examples

NR: gawk '{print NR "-" \$1}' mobile.txt

```

linux@ubuntu:~/files$ gawk '{print NR "-" $1}' mobile.txt
1-Deepak
2-Sunil
3-Aman
4-Kuldeep
5-Sundip
linux@ubuntu:~/files$

```

- **RS:** gawk 'BEGIN{FS=":"; RS="-"} {print \$1, \$6, \$7}' /etc/passwd

```
linux@ubuntu:~/files$ gawk 'BEGIN{FS=":"; RS="-"} {print $1,$6,$7}' /etc/passwd
root /root /bin/bash
daemon
data
data 34 backup
Reporting System (admin) 65534 nobody
network /run/systemd/netif /usr/sbin/nologin
systemd
resolve /run/systemd/resolve /usr/sbin/nologin
syslog
autoipd /var/lib/avahi
autoipd usbmux daemon,, /var/lib/usbmux
pk
helper
pk
helper service,,
pk
helper
dispatcher /var/run/speech
dispatcher /nonexistent
daemon colord colour management daemon,, /var/lib/colord
initial
setup /run/gnome
initial
setup/ Gnome Display Manager /var/lib/gdm3
```

- **OFS:** gawk 'BEGIN{FS=":"; OFS="-"} {print \$1, \$6, \$7}' /etc/passwd

```
linux@ubuntu:~/files$ gawk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
root-/root-/bin/bash
daemon-/usr/sbin-/usr/sbin/nologin
bin-/bin-/usr/sbin/nologin
sys-/dev-/usr/sbin/nologin
sync-/bin-/bin/sync
games-/usr/games-/usr/sbin/nologin
man-/var/cache/man-/usr/sbin/nologin
lp-/var/spool/lpd-/usr/sbin/nologin
mail-/var/mail-/usr/sbin/nologin
news-/var/spool/news-/usr/sbin/nologin
uucp-/var/spool/uucp-/usr/sbin/nologin
proxy-/bin-/usr/sbin/nologin
www-data-/var/www-/usr/sbin/nologin
backup-/var/backups-/usr/sbin/nologin
list-/var/list-/usr/sbin/nologin
irc-/var/run/ircd-/usr/sbin/nologin
gnats-/var/lib/gnats-/usr/sbin/nologin
nobody-/nonexistent-/usr/sbin/nologin
systemd-network-/run/systemd/netif-/usr/sbin/nologin
systemd-resolve-/run/systemd/resolve-/usr/sbin/nologin
syslog-/home/syslog-/usr/sbin/nologin
messagebus-/nonexistent-/usr/sbin/nologin
_apt-/nonexistent-/usr/sbin/nologin
uuidd-/run/uuidd-/usr/sbin/nologin
avahi-autoipd-/var/lib/avahi-autoipd-/usr/sbin/nologin
usbmux-/var/lib/usbmux-/usr/sbin/nologin
dnsmasq-/var/lib/misc-/usr/sbin/nologin
rtkit-/proc-/usr/sbin/nologin
cups-pk-helper-/home/cups-pk-helper-/usr/sbin/nologin
speech-dispatcher-/var/run/speech-dispatcher-/bin/false
whoopsie-/nonexistent-/bin/false
kernoops-/-/usr/sbin/nologin
saned-/var/lib/saned-/usr/sbin/nologin
pulse-/var/run/pulse-/usr/sbin/nologin
avahi-/var/run/avahi-daemon-/usr/sbin/nologin
colord-/var/lib/colord-/usr/sbin/nologin
hplip-/var/run/hplip-/bin/false
```


Exploring sed Editor basics

The Sed command is a powerful tool in the Linux and Unix operating systems that is used for streamlining text processing. Sed, short for Stream EDitor, can be used to search, delete, insert, or replace characters within a file or multiple files with minimal effort.

Basic 'sed' Syntax:

The basic syntax for 'sed' is as follows:

Example

```
sed [options] 'command' inputfile
```

'Options': These are optional and can modify the behavior of 'sed'. Some common options include '-i' (in-place edit), '-n' (suppress automatic printing), and others.

'Command': This is the 'sed' command you want to execute.

'Inputfile': This is the file you want to process. If you omit it, 'sed' will read from standard input.

Common 'sed' Commands

Search and Replace: The 's' (substitute) command is used to search for a pattern and replace it with another.

Code: sed 's/old/new/' inputfile

- This command replaces the first occurrence of "old" with "new" on each line.

Global Search and Replace

- To replace all occurrences of a pattern on each line, use the 'g' flag.

Code: sed 's/old/new/g' inputfile

Using Regular Expressions:

- We can use regular expressions in 'sed' to match more complex patterns. For example, to replace "apple" or "apples" with "fruit":

Code: sed 's/apple[s]*\$/fruit/g' inputfile

Delete Lines

- The 'd' command is used to delete lines that match a pattern:

Code: sed '/pattern/d' inputfile

Print Specific Lines

- Use the '-n' option to suppress automatic printing and the 'p' command to print specific lines

Code: sed -n '5,10p' inputfile

Multiple Commands

- You can combine multiple 'sed' commands using semicolons. For example, replacing and then deleting lines.

Code: `sed -e 's/old/new/' -e '/pattern/d' inputfile`

In-Place Editing

To edit a file in-place (i.e., save changes to the original file), you can use the '-i' option. Be cautious when using this option, as it will overwrite the input file. Always make a backup before using it.

Code: `sed -i 's/old/new/' inputfile`

A few examples of common 'sed' operations

- Replace all occurrences of "cat" with "dog" in a file and save the changes:

Code: `sed -i 's/cat/dog/g' myfile.txt`

- Delete lines containing the word "apple" from a file:

Code: `sed -i '/apple/d' myfile.txt`

- Print lines between lines that match "start" and "end":

Code: `sed -n '/start/,/end/p' myfile.txt`

Let us sum up

The sed editor writes to a destination file only the data lines that contain the text pattern.

The case command should call the appropriate function according to the character selection expected from the menu. It's always a good idea to use the default case command character (the asterisk) to catch any incorrect menu entries.

Check your progress

1. What is the correct syntax for defining a function in bash?*

A) function name { ... }

B) def name { ... }

C) func name { ... }

D) function name = { ... }

2. How do you return a value from a function in bash?*

A) return value

B) exit value

C) output value

D) echo value

3. How do you access a variable defined outside a function inside the function in bash?*

- A) By using the global keyword
- B) By using the extern keyword
- C) Variables are automatically accessible
- D) By passing the variable as a parameter

4. How do you define an array in bash?*

- A) array = (value1 value2 value3)
- B) array = [value1, value2, value3]
- C) array = {value1, value2, value3}
- D) array = <value1, value2, value3>

5. Which statement is true about recursion in bash functions?*

- A) Bash does not support recursion.
- B) Recursion is supported but can lead to a stack overflow if not managed carefully.
- C) Recursion is the only way to loop in bash.
- D) Recursion is faster than iteration in bash.

6. How do you include a library file in a bash script?*

- A) include "library.sh"
- B) source library.sh
- C) import library.sh
- D) use library.sh

7. Can you define and use functions directly from the bash command line?*

- A) Yes
- B) No

8. Which command is commonly used to create text menus in a bash script?*

- A) menu
- B) select
- C) choose
- D) options

9. Which tool can be used to create graphical dialogs in bash scripts?*

- A) dialog
- B) zenity
- C) yad
- D) All of the above

10. Which toolkit is commonly used for adding X Window graphics in bash scripts?*

- A) GTK B) QT C) Xlib D) All of the above

11. What is the primary use of the sed command in bash?*

- A) Text editing B) File compression
C) Network management D) Package installation

12. What does gawk stand for?*

- A) GNU AWK B) General AWK C) Global AWK D) Graphical AWK

13. How do you use sed to replace the first occurrence of "foo" with "bar" in a file?*

- A) sed 's/foo/bar/' filename B) sed 'r/foo/bar/' filename
C) sed 'c/foo/bar/' filename D) sed 't/foo/bar/' filename

Here are the answers:

1. A) 2. A) 3. C) 4. A) 5. B) 6. B) 7. A) 8. B) 9. D)
10. D) 11. A) 12. A) 13. A)

Self assessment questions

1. How to defining sed editor command in command line. Give an example.
2. Difference between Array and variable function.
3. What is function recursion with example?
4. How to creating a library and using functions on command line.
5. Explain in detailed about sed and gawk commands with examples.
6. Write a shell script program to create a following GUI tools.
 - Creating text menu
 - Building text window widget
7. How to use edit command at the sed editor.

8. Write about functions and arrays in Linux.
9. Explain about adding color to scripts layout and the functions.

Open source e-content links

<https://www.tutorialspoint.com/unix/unix-shell-functions.htm>

<https://www.geeksforgeeks.org/function-command-in-linux-with-examples/>

<https://www.geeksforgeeks.org/introduction-to-graphical-user-interface-of-redhat-linux-operating-system/>

Glossary

sed is a powerful stream editor used for filtering and transforming text.

sed: The command itself to invoke the stream editor.

Usage: sed [options] 'script' [file]

s/pattern/replacement/: Substitutes replacement for pattern.

Usage: sed 's/old/new/' file

g: Applies substitution globally on each line (i.e., all occurrences).

Usage: sed 's/old/new/g' file

ì\ : Inserts a line before the matched pattern.

Usage: sed '/pattern/ì\text to insert' file

a\ : Appends a line after the matched pattern.

Usage: sed '/pattern/a\text to append' file

d: Deletes lines that match a pattern.

Usage: sed '/pattern/d' file

p: Prints lines that match a pattern.

Usage: sed -n '/pattern/p' file

-e: Allows multiple commands to be executed.

Usage: sed -e 'command1' -e 'command2' file

-n: Suppresses automatic printing of pattern space (use with p for controlled output).

Usage: sed -n 'p' file

-f: Reads sed commands from a file.

awk is a powerful programming language for pattern scanning and processing.

awk: The command to invoke the AWK programming language.

Usage: awk [options] 'program' [file]

{ }: Denotes an action block in an awk program.

Usage: awk '{print \$1}' file

\$n: Represents the nth field in a record (default delimiter is whitespace).

Usage: awk '{print \$1, \$3}' file

BEGIN: Block executed before any input is processed.

Usage: awk 'BEGIN {print "Start"} {print \$1}' file

END: Block executed after all input is processed.

Usage: awk '{print \$1} END {print "End"}' file

/pattern/: Selects records that match a pattern.

Usage: awk '/pattern/ {print \$1}' file

print: Outputs text or variables.

Usage: awk '{print \$1, \$2}' file

FS: Field Separator, sets the delimiter for fields.

Usage: awk 'BEGIN {FS=","} {print \$1}' file

OFS: Output Field Separator, sets the delimiter for output fields.

Usage: `awk 'BEGIN {OFS="\t"} {print $1, $2}' file`

NR: Built-in variable representing the current record number.

Usage: `awk '{print NR, $0}' file`

NF: Built-in variable representing the number of fields in the current record.

e-books

1. "Linux Desktop Hacks: Tips & Tools for Customizing and Optimizing Your Desktop" by Kyle Rankin

2. "KDE 4.0: Using the KDE Desktop Environment" by David F. Swersky

UNIT – IV

Objectives:

- Utilize regular expressions (regex) to perform complex pattern matching, search, and text manipulation tasks efficiently.
- Master advanced sed features to perform sophisticated text processing tasks, such as multi-line editing, complex substitutions, and script-based manipulations.
- Leverage advanced features of gawk to perform sophisticated data processing, reporting, and text manipulation tasks.

4.1 Regular Expressions

Regexps are acronyms for regular expressions. Regular expressions are special characters or sets of characters that help us to search for data and match the complex pattern. Regexps are most commonly used with the Linux commands:- grep, sed, tr, vi.

| S no | Symbol | Description |
|------|--------|--|
| 1 | . | It is called a wild card character; It matches any one character other than the new line. |
| 2 | ^ | It matches the start of the string. |
| 3 | \$ | It matches the end of the string. |
| 4 | * | It matches up to zero or more occurrences i.e. any number of times of the character of the string. |
| 5 | \ | It is used for escape following character. |
| 6 | () | It is used to match or search for a set of regular expressions. |
| 7 | ? | It matches exactly one character in the string or stream. |

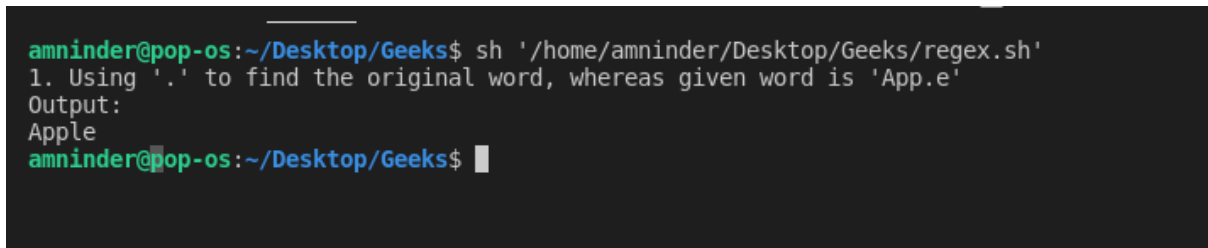
✚ Using “.” (dot) to match strings.

Using “.” we can find a string if we do not know the exact string, or we just remember only the start and end of the string, we can use “.” As a missing character, and it will fill that missing character. Let’s see an example for better understanding:’ File contains the fruit’s name, and we are going to use regular expressions on this file.

Script

```
#!/bin/sh
# Basic Regular Expression
# 1. Using “.” to match strings.
# loading the text file fruits_file=`cat fruit.txt | grep App.e`
# here the original (answer) word will be Apple,
# but because we don't know the spelling of the Apple,
# will put a dot (.) in that place.
echo "1. Using ‘.’ to find the original word, whereas given word is ‘App.e’"
# displaying output
echo "Output:"
echo "$fruits_file"
```

Output



```
amninder@pop-os:~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'
1. Using '.' to find the original word, whereas given word is 'App.e'
Output:
Apple
amninder@pop-os:~/Desktop/Geeks$
```

✚ Using “^” (caret) to match the beginning of the string

Using “^”, we can find all the strings that start with the given character. Example for a better understanding. Here we are trying to find all the fruit names that start with the letter B:

Script

```
#!/bin/sh
# Basic Regular Expression
# 2. Using “^” (caret) to match the beginning of the string
# loading the text file
```

```
fruits_file=`cat fruit.txt | grep ^B`
```

```
echo "2. Using '^' to find out all the words that start with the letter 'B'"  
# displaying output  
echo "Output:"  
echo "$fruits_file"
```

Output

```
amninder@pop-os:~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'  
2. Using '^' to find out all the words that start with the letter 'B'  
Output:  
Banana  
Bilberry  
Blackberry  
Blackcurrant  
Blueberry  
Boysenberry  
Blood orange  
amninder@pop-os:~/Desktop/Geeks$
```

✚ Using "\$" (dollar sign) to match the ending of the string

Using "\$" we can find all the strings that end with the given character. Example for a better understanding. Here we are trying to find all the fruit's names that end with the letter e:

Script

```
#!/bin/sh  
# Basic Regular Expression  
# 3. Using "$" (dollar) to match the ending of the string  
# loading the text file  
fruits_file=`cat fruit.txt | grep e$`  
echo "3. Using '$' to find out all the words that end with the letter 'e'"  
# displaying output  
echo "Output:"  
echo "$fruits_file"
```

Output

```
amninder@pop-os: ~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'
3. Using '$' to find out all the words that end with the letter 'e'
Output:
Apple
Custard apple
Date
Grape
Jujube
Lime
Lychee
Cantaloupe
Nectarine
Nance
Olive
Orange
Blood orange
Clementine
Mandarine
Tangerine
Prune
Pineapple
Pomegranate
Quince
Solanum quitoense
amninder@pop-os:~/Desktop/Geeks$
```

✚ Using “*” (an asterisk) to find any number of repetitions of a string

Using “*”, we can match up to zero or more occurrences of the character of the string. Example for a better understanding. Here we are trying to find all the fruit’s names that

has one or more occurrences of ‘ap’ one after another in it.

Script

```
#!/bin/sh
# Basic Regular Expression
# 4. Using “*” to find any number of repetition of a string
# loading the text file
fruits_file=`cat fruit.txt | grep ap*le`
echo “4. Using ‘*’ to find out all the fruits name that has one or more
occurrence of ‘ap’ one after another in it”
# displaying output
echo “Output:”
echo “$fruits_file”
```

Output

```
amninder@pop-os:~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'
4. Using '*' to find out all the fruits name that has one or more occurrence
of 'ap' one after another in it
Output:
Custard apple
Pineapple
amninder@pop-os:~/Desktop/Geeks$
```

✚ Using “\” (a backslash) to match the special symbol

Using “\” with special symbols like whitespace (“ ”), newline(“\n”), we can find strings from the file. Example for a better understanding. Here we are trying to find all the fruit’s names that have space in their full names.

Script

```
#!/bin/sh
# Basic Regular Expression
# 5. Using “\” to match the special symbol
# loading the text file
fruits_file=`cat fruit.txt | grep “\ “`
echo “5. Using ‘\’ to find out all the fruits name that has single space in
their full name”
# displaying output
echo “Output:”
echo “$fruits_file”
```

Output

```
amninder@pop-os:~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'
5. Using '\ ' to find out all the fruits name that has single space in their
full name
Output:
Chico fruit
Custard apple
Goji berry
Juniper berry
Miracle fruit
Blood orange
Purple mangosteen
Salal berry
Star fruit
Solanum quitoense
Ugli fruit
amninder@pop-os:~/Desktop/Geeks$
```

✚ Using “()” (braces) to match the group of regexp.

Using “()”, we can find matched strings with the pattern in the “()”. Example for a better understanding. Here we are trying to find all the fruit’s names that have space in their full name.

Script

```
#!/bin/sh
# Basic Regular Expression
# 6. Using “()” (braces) to match the group of regexp.
# loading the text file
fruits_file=`cat fruit.txt | grep -E “(fruit)”`
echo “6. Using ‘()’ to find out all the fruits name that has word ‘fruit’ in it”
# displaying output
echo “Output:”
echo “$fruits_file”
```

Output

```
amninder@pop-os:~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'
6. Using '()' to find out all the fruits name that has word 'fruit' in it
Output:
Chico fruit
Dragonfruit
Grapefruit
Jackfruit
Kiwifruit
Miracle fruit
Passionfruit
Star fruit
Ugli fruit
amninder@pop-os:~/Desktop/Geeks$ █
```

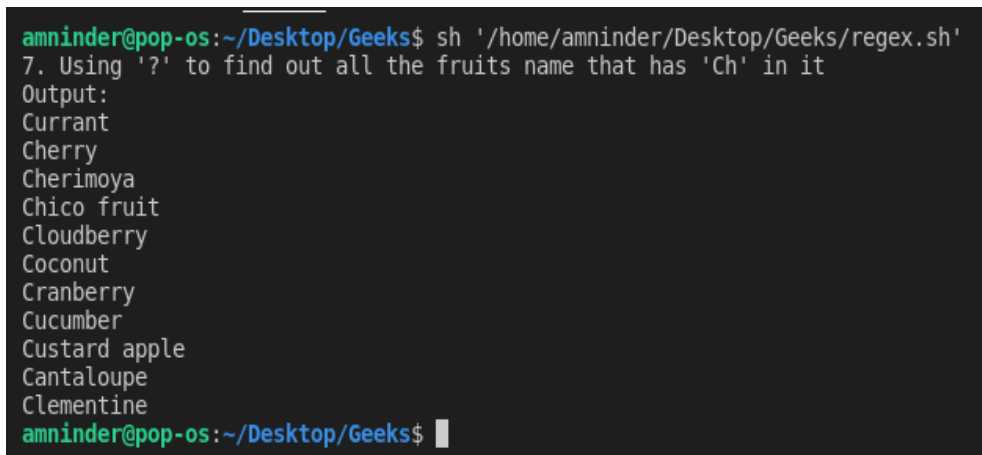
Using “?”(question mark) to find all the matching characters

Using “?”, we can match 0 or 1 repetitions of the preceding. It will match either ‘a’ or ‘ab’. Example for better understanding. Here we are trying to find all the fruit’s names that have the character ‘Ch’ in them.

Script

```
#!/bin/sh
# Basic Regular Expression
# 7. Using "?"(question mark) to match the
# loading the text file
fruits_file=`cat fruit.txt | grep -E Ch?`
echo "7. Using '?' to find out all the fruits name that has 'Ch' in it"
# displaying output
echo "Output:"
echo "$fruits_file"
```

Output



```
amninder@pop-os:~/Desktop/Geeks$ sh '/home/amninder/Desktop/Geeks/regex.sh'
7. Using '?' to find out all the fruits name that has 'Ch' in it
Output:
Currant
Cherry
Cherimoya
Chico fruit
Cloudberry
Coconut
Cranberry
Cucumber
Custard apple
Cantaloupe
Clementine
amninder@pop-os:~/Desktop/Geeks$
```

Extending our patterns

❖ **Wildcards:** Wildcards are characters used to match patterns in filenames.

Common wildcards include:

- ***** (asterisk): Matches any sequence of characters.
- **?** (question mark): Matches any single character.
- **[]** (square brackets): Matches any character within the specified range or set.

Example

To match all files with a **.txt** extension in a directory, we can use the wildcard ***.txt**.

-
- ❖ **Regular Expressions:** Regular expressions (regex or regexp) are powerful tools for pattern matching in text. We can use them with tools like **grep**, **sed**, and **awk**.
 - **grep** can be used to search for patterns within files or text streams. **Script:**
grep 'pattern' file.txt
 - **sed** can be used to search and replace patterns in text
Script: sed 's/pattern/replacement/g' file.txt
 - **awk** is a versatile text processing tool that can be used to work with structured data. It supports regular expressions for pattern matching.
 - ❖ **Brace Expansion:** Brace expansion is used to generate strings by specifying a range or list of values inside curly braces. For example, you can generate a list of file names with different extensions. **Script:** echo file.{txt,md,csv}
 - ❖ **Extended Globbing:** Some shells, such as Bash, support extended globbing patterns for more advanced matching. You can enable this feature using **shopt -s extglob**. With extended globbing, you can do things like matching files based on specific patterns or excluding certain patterns. For example, to list all files except those ending in **.bak**: **Script:** shopt -s extglob ls !(*.bak)
 - ❖ **Combining Patterns:** You can combine patterns using logical operators, such as **&&** (and) and **||** (or), to create more complex matching conditions. For example, you can list files that are either **.txt** or **.md** files: **Script:** ls *.txt || ls *.md.
 - ❖ **Parameter Expansion:** Shell scripts also allow for parameter expansion, which can be used to manipulate variables and strings. For example, you can extract a substring from a variable. **Script:** text="Hello, World" echo \${text:0:5}

Output

"Hello"

- ❖ **Custom Functions:** In shell scripting, you can create custom functions to extend your pattern matching and manipulation capabilities. These functions can be used to encapsulate complex logic and make your scripts more modular and readable.

Creating expressions

The **expr** command in Unix evaluates a given expression and displays its corresponding output. It is used for:

- Basic operations like addition, subtraction, multiplication, division, and modulus on integers.
- Evaluating regular expressions, string operations like substring, length of strings etc.

Syntax: `$expr expression`

Using expr for basic arithmetic operations:

Example: Addition

```
$expr 12 + 8
```

Example: Multiplication

```
$expr 12 \* 2
```

Output

A terminal window titled 'root@genesis101: ~' with a menu bar (File, Edit, View, Search, Terminal, Help) and a dragon logo in the background. The terminal shows the following commands and outputs:

```
root@genesis101:~# expr 12 + 8
20
root@genesis101:~# expr 12 \* 2
24
root@genesis101:~#
```

Performing operations on variables inside a shell script

***Example:** Adding two numbers in a script*

```
echo "Enter two numbers"
```

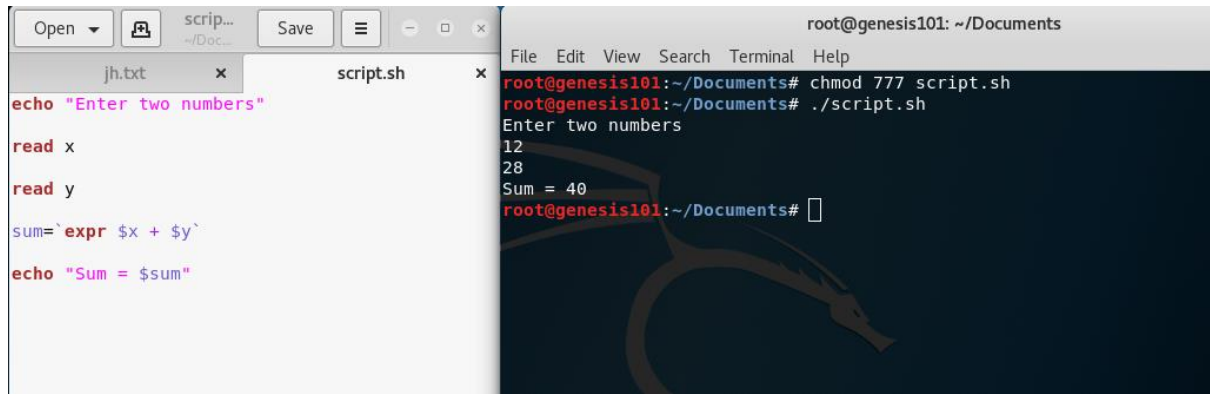
```
read x
```

```
read y
```

```
sum=`expr $x + $y`
```

```
echo "Sum = $sum"
```


Output



```
Open  scrip...  Save  jh.txt  script.sh  root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# chmod 777 script.sh
root@genesis101:~/Documents# ./script.sh
Enter two numbers
12
28
Sum = 40
root@genesis101:~/Documents#
```

Comparing two expressions

Example

```
x=10
```

```
y=20
```

```
# matching numbers with '='
```

```
res=`expr $x = $y`
```

```
echo $res
```

```
# displays 1 when arg1 is less than arg2
```

```
res=`expr $x \< $y`
```

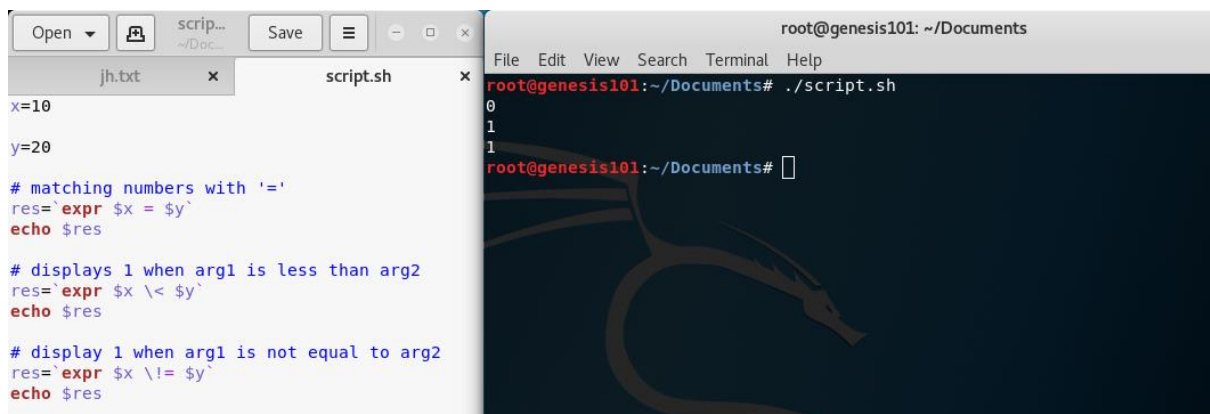
```
echo $res
```

```
# display 1 when arg1 is not equal to arg2
```

```
res=`expr $x \!= $y`
```

```
echo $res
```

Output



```
Open  scrip...  Save  jh.txt  script.sh  root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
0
1
1
root@genesis101:~/Documents#
```

For String operations

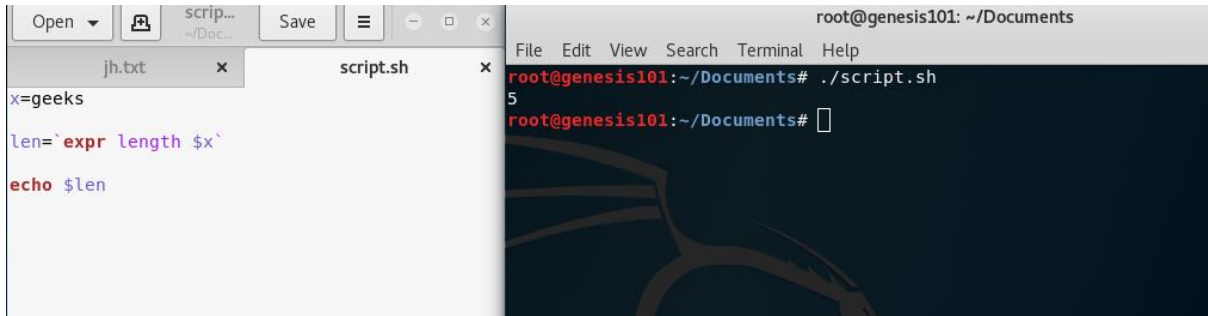
Example: Finding length of a string

```
x=geeks
```

```
len=`expr length $x`
```

```
echo $len
```

Output



The screenshot shows a terminal window with a file editor on the left and a terminal on the right. The file editor shows a script named 'script.sh' with the following content:

```
x=geeks
len=`expr length $x`
echo $len
```

The terminal window shows the execution of the script:

```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
5
root@genesis101:~/Documents#
```

Finding substring of a string

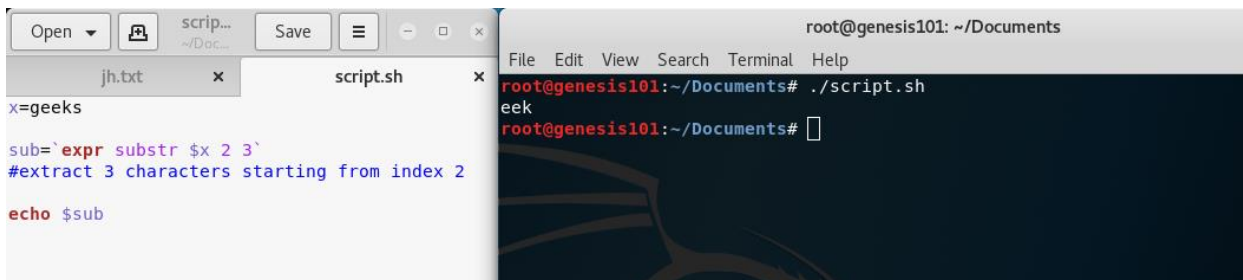
```
x=geeks
```

```
sub=`expr substr $x 2 3`
```

#extract 3 characters starting from index 2

```
echo $sub
```

Output



The screenshot shows a terminal window with a file editor on the left and a terminal on the right. The file editor shows a script named 'script.sh' with the following content:

```
x=geeks
sub=`expr substr $x 2 3`
#extract 3 characters starting from index 2
echo $sub
```

The terminal window shows the execution of the script:

```
root@genesis101: ~/Documents
File Edit View Search Terminal Help
root@genesis101:~/Documents# ./script.sh
eek
root@genesis101:~/Documents#
```

4.2 Advanced sed

1. **Search and Replace:** **sed** is commonly used for search and replace operations using these command, we can replace occurrences of a pattern with another string.

Script: `sed 's/old_pattern/new_pattern/g' input.txt`

2. **Using Regular Expressions:** **sed** supports regular expressions for pattern matching. This allows you to perform complex substitutions. Example: to replace all email addresses in a file with "EMAIL_HIDDEN":

Script: `sed 's/[A-Za-z0-9._%+-]\+@[A-Za-z0-9.-]\+\.[A-Za-z]\{2,\}/EMAIL_HIDDEN/g' input.txt`

3. **In-Place Editing:** To edit a file in-place, use the **-i** option. This allows you to save the changes back to the original file:

Script: `sed -i 's/old_pattern/new_pattern/g' input.txt`

4. **Deleting Lines:** **sed** can delete lines that match a pattern. For example, to delete all lines containing the word "DELETE":

Script: `sed '/DELETE/d' input.txt`

5. **Inserting and Appending Text:** An insert or append text before or after specific lines. For example, to add a line at the beginning of a file:

Script: `sed '1i This is the first line' input.txt`

6. **Using Capture Groups:** Capture groups allow you to extract and rearrange parts of the matched pattern. For example, to swap first and last names in a list:

Script: `sed 's/(.*) (.*)/2 \1/g' names.txt`

7. **Multiple Operations:** Chain multiple **sed** operations together by separating them with semicolons. For instance, to replace multiple patterns in one pass:

Script: `sed -e 's/pattern1/replacement1/g' -e 's/pattern2/replacement2/g' input.txt`

-
8. **Using Variables:** Use shell variables within your **sed** commands, making your scripts more dynamic. For example, using a variable to store the replacement text:

Script: replacement="new_value" sed "s/old_value/\$replacement/g" input.txt

9. **Conditional Editing:** Conditionally edit lines based on patterns or line numbers. For example, to replace a line only if it matches a specific pattern:

Script: sed '/pattern/s/old/new/' input.txt

10. **Script Files:** For complex operations, you can store **sed** commands in a separate script file and execute it with the **-f** option:

Script: sed -f myscript.sed input.txt

11. **Escape Special Characters:** If you need to match or replace special characters (e.g., **/**, **&**, or **^**), make sure to escape them with backslashes in your **sed** command.

Using multiline commands

Use a single ampersand (&) or two ampersands (&&) to separate multiple commands on one command line.

There are 3 ways to run multiple shell commands in one line:

- ❖ Use ;

No matter the first command cmd1 run successfully or not, always run the second command cmd2:

Script:

```
# cmd1; cmd2
```

```
$ cd myfolder; ls # no matter cd to myfolder successfully, run ls
```

- ❖ Use &&

Only when the first command cmd1 run successfully, run the second command cmd2:

Script:

```
# cmd1 && cmd2
```

```
$ cd myfolder && ls # run ls only after cd to myfolder
```

❖ Use ||

Only when the first command cmd1 failed to run, run the second command cmd2:

Script:

```
# cmd1 || cmd2
```

```
$ cd myfolder || ls # if failed cd to myfolder, `ls` will run
```

Understanding the hold space

The Hold command puts a newline in the hold space and then appends the current line to the hold space. Even when the hold space is empty, the Hold command places a newline before the contents of the pattern space. The exchange command (x) swaps the contents of the hold space and the pattern space.

✚ Sed has two types of internal storage space:

- **Pattern space:** In which is used as part of the typical sed execution flow. Pattern space is the internal sed buffer where sed places, and modifies, the line it reads from the input file.
- **Hold space:** This is an additional buffer available where sed can hold temporary data. Sed allows you to move data back and forth between pattern space and hold space, but you cannot execute the typical sed commands on the hold space. Pattern space gets deleted at the end of every cycle in a typical sed execution flow. However, the content of the hold space will be retained from one cycle to the next; it is not deleted between cycles.

To create a new text file to be used for the sed hold space examples:

Example

```
$ vi empname.txt
```

```
John
```

```
CEO
```

```
Jason Smith
```

```
IT Manager
```

```
Raj Reddy
```

```
Sysadmin
```

```
Anand Ram
```

- ❖ **'H' command:** We can use the 'H' command to copy the current pattern space (line) into the hold space.

Example

```
sed 'h' file.txt
```

- ❖ **'h' command:** The 'h' command appends the current pattern space to the hold space, separated by a newline character. This can be useful for accumulating lines in the hold space.

Example

```
sed 'H' file.txt
```

- ❖ **'g' command:** To copy the contents of the hold space back into the pattern space, we can use the 'g' command. This allows you to retrieve the stored text and work with it in your script.

Example

```
sed 'h;g' file.txt
```

- ❖ **'g' command:** The 'G' command appends the contents of the hold space to the pattern space, separated by a newline character. This is useful when you want to combine the stored text with the current line.

Example

```
sed 'h;G' file.txt
```

- ❖ **Clearing the hold space:** We can clear the hold space using the 'x' command.

Example

```
sed 'h;x' file.txt
```

Negating a command

In Bash, you can negate a condition using the "!" operator. We can negate the result of any command or condition with the "!" operator. It's also worth noting that you can use [[...]] instead of [...] for string and numeric comparison.

Example

```
if [ ! "$x" -eq 5 ]
```

```
then
```

```
    echo "x is not equal to 5"
```

- You can also negate a string comparison using the "!" –

```
if [ ! "$s" = "hello" ]  
then  
echo "s is not equal to hello"  
fi
```

You can negate the result of any command or condition with the "!" operator.

Example

```
if [[ "$x" != 5 ]]  
then  
echo "x is not equal to 5"  
fi
```

It's also worth noting that you can use `[[...]]` instead of `[...]` for string and numeric comparison.

Changing the flow

- ❖ **Conditional Statements (if-else-fi):** Conditional statements are used to make decisions in your script based on specific conditions.

Example

```
if [ $num -gt 10 ]; then  
echo "Number is greater than 10"  
else  
echo "Number is not greater than 10"  
fi
```

- ❖ **Case Statements (case-esac):** Case statements are used when you have multiple conditions to test against a variable. It is similar to a switch statement in other programming languages.

Example

```
case $choice in  
"1")  
echo "You chose option 1"  
;;  
"2")
```

```
echo "You chose option 2"
```

```
;;  
*)  
echo "Invalid choice"  
;;
```

- ✚ **Loops (for, while, until):** Loops are used for repetitive tasks. we can use **'for'**, **'while'**, and **'until'** loops to execute a block of commands multiple times.

- ❖ **For Loop**

- Example**

- ```
for var in list; do
commands to execute for each item in the list
Done.
```

- ❖ **While Loop**

- Example**

- ```
while [ condition ]; do  
# commands to execute as long as the condition is true  
Done
```

- ❖ **Until Loop**

- Example**

- ```
until [condition]; do
commands to execute until the condition becomes true
Done
```

- ❖ **Break and Continue:** Inside loops, you can use **'break'** to exit the loop prematurely and **'continue'** to skip the current iteration and move to the next one.

- Example**

- ```
for i in {1..10}; do  
if [ $i -eq 5 ]; then  
break # exit the loop when i is 5  
fi  
echo "Iteration: $i"  
done
```


Replacing via a pattern

- ❖ **Using 'sed'**: 'sed' (stream editor) is a powerful tool for text manipulation, including text replacement using patterns.
- **'pattern'**: This is the regular expression pattern you want to search for in the input.
- **'replacement'**: This is the text you want to replace the matched pattern with.
- **'g'**: This is an optional flag that tells **'sed'** to replace all occurrences of the pattern in each line. If you omit it, **'sed'** will replace only the first occurrence in each line.

Example of using 'sed' to replace "oldtext" with "newtext" in a file:

```
sed 's/oldtext/newtext/g' input.txt > output.txt
```

- To edit the file in-place without creating a new file, you can use the **'-i'** flag:

```
sed -i 's/oldtext/newtext/g' input.txt
```

- ❖ **Using 'awk'**: awk is another versatile tool for text manipulation, and it can be used to replace text based on patterns.

Syntax

```
awk '{gsub(/pattern/, "replacement")}'1' inputfile > outputfile
```

- **'pattern'**: This is the regular expression pattern you want to search for.
- **'replacement'**: This is the text you want to replace the matched pattern with.

Example of using **'awk'** to replace "oldtext" with "newtext" in a file:

```
awk '{gsub(/oldtext/, "newtext")}'1' input.txt > output.txt
```

- ❖ **Using Shell String Manipulation:**

- For simple replacements within a shell script, we can use parameter expansion

Example

```
string="This is the old text"  
newstring="${string/old text/new text}"  
echo "$newstring"
```

Using sed in Scripts

The Linux sed command is most commonly used for substituting text. It searches for the specified pattern in a file and replaces it with the wanted string. To replace text using sed, use the substitute command s and delimiters (in most cases, slashes - /) for separating text fields.

```
soflja@soflja-VirtualBox:~$ sed 's/box/bin/' foxinbox.txt
Knox in bin.
Fox in socks.

Knox on fox in socks in bin.
Socks on Knox and Knox in bin.

Fox in socks on bin on Knox.
```

Creating sed utilities

SED command in UNIX stands for stream editor and it can perform lots of functions on file like searching, find and replace, insertion or deletion. Though most common use of SED command in UNIX is for substitution or for find and replace.

Here's a general outline of how to create a custom '**sed**' utility:

- ❖ **Define the pattern to match:** Start by defining the regular expression pattern that you want to match in the input text.
- ❖ **Specify the replacement text:** Determine the text that you want to replace the matched pattern with.
- ❖ **Use s command:** Within your custom '**sed**' script, use the **s** command to perform the substitution.

Syntax: sed -e 's/pattern/replacement/g' inputfile

- '**pattern**': This is the regular expression pattern to match.
- '**replacement**': This is the text with which you want to replace the matched pattern.
- '**g**': An optional flag to replace all occurrences in each line. Omitting it replaces only the first occurrence in each line.
- **Execute the custom sed script:** Run your '**sed**' script with the '**-e**' option and provide the input file you want to process.
-

Example of creating a custom '**sed**' utility that replaces all occurrences of "apple" with "banana" in a text file.

```
sed -e 's/apple/banana/g' input.txt
```

Creating More Complex sed Utilities

For more complex **'sed'** utilities, you can combine multiple **'sed'** commands and use control flow constructs. For example, you can use **'if'** conditions and **'b'** (branch) commands to perform different substitutions based on conditions.

Example of a custom **'sed'** utility that replaces "apple" with "banana" but only if the line starts with "fruit":

```
sed -e '/^fruit:/s/apple/banana/g' input.txt
```

4.3 Advanced gawk

- ❖ **Regular Expressions:** Gawk, like Awk, supports regular expressions. You can use regular expressions to pattern match and extract specific data from text files.

Example: `gawk '/pattern/ { print $2 }' file.txt`

- ❖ **Advanced Field Separators:** Gawk allows you to set custom field separators with the **-F** option. This is useful when working with files that have delimiters other than spaces or tabs. For instance, to process a file with a semicolon delimiter:

Example: `gawk -F ';' '{ print $2 }' file.txt`

- ❖ **Built-in Functions:** Gawk provides numerous built-in functions for text and numeric operations. You can use functions like **split()**, **substr()**, **length()**, and **sprintf()** to manipulate data. For example, you can split a field into an array:

Example: `gawk '{ split($3, arr, "-"); print arr[2] }' file.txt`

- ❖ **User-Defined Functions:** Gawk allows you to create your own functions for custom data processing. This can make your scripts more modular and maintainable. Here's an example of defining and using a custom function:

Example: `gawk 'function myfunc(x) { return x * 2 } { print myfunc($1) }'`
file.txt

- ❖ **Arrays:** Gawk supports arrays, which are useful for storing and processing data. You can use arrays to aggregate and analyze data from a file:

Example: `gawk '{ count[$1]++ } END { for (key in count) print key, count[key] }' file.txt`

-
- ❖ **Control Structures:** Gawk supports control structures like **if-else** and **while** loops, allowing you to perform conditional operations and iterative tasks within your scripts.

 - ❖ **Output Formatting:** Gawk offers extensive control over the formatting of output. You can use the **printf** function to format and display data in a specific way.
Example: `gawk '{ printf("Name: %s, Age: %d\n", $1, $2) }' file.txt`

 - ❖ **Reading Multiple Files:** Gawk can process multiple files and perform actions separately on each file. Use the **ARGIND** variable to identify the current file being processed.
Example: `gawk '{ print "File:", ARGIND, "Line:", NR, $0 }' file1.txt file2.txt`

 - ❖ **Advanced Data Processing:** Gawk is capable of more advanced data processing tasks, such as calculating statistics, parsing structured data formats, and generating reports.

 - ❖ **Error Handling:** Gawk provides error handling mechanisms, allowing you to handle and report errors gracefully within your scripts.

Reexamining gawk

- ❖ **Built-in Variables:** Gawk provides a variety of built-in variables to help you work with your data:
 - **NF** represents the number of fields in the current record.
 - **NR** indicates the current record number.
 - **FS** is the input field separator (default is whitespace).
 - **OFS** is the output field separator (used when printing fields).
 - **RS** is the input record separator (default is a newline).
 - **ORS** is the output record separator (used when printing records).

- ❖ **Pattern Matching:** Gawk uses regular expressions for pattern matching, allowing you to search for specific text patterns within your data. we can use regular expressions to match and process data selectively.

-
- ❖ **Control Structures:** Gawk supports conditionals (**'if', 'else', 'else if'**) and loops (**'while, for'**) for controlling the flow of your data processing. We can use these control structures to perform complex logic within your scripts.

 - ❖ **Functions:** Gawk provides a rich set of built-in functions for both text and numeric operations.
 - **gsub():** Global substitution of text.
 - **index():** Find the position of a substring in a string.
 - **length():** Determine the length of a string.
 - **split():** Split a string into an array based on a delimiter.
 - **substr():** Extract a substring from a string.
 - **system():** Run shell commands from within an awk script.

 - ❖ **User-Defined Functions:** We can define your own custom functions in Gawk, which makes your scripts more modular and easier to maintain.

 - ❖ **Arrays:** Gawk supports arrays, both indexed and associative. This is useful for aggregating data, creating data structures, and performing more advanced data processing tasks.

 - ❖ **BEGIN and END Blocks:** Gawk allows you to specify actions to be taken before processing begins (**BEGIN** block) and after processing is complete (**END** block). These blocks are typically used for setup and cleanup tasks.

 - ❖ **Multi-File Processing:** You can process multiple input files in a single Gawk command, and Gawk keeps track of the current file being processed using the **'ARGIND'** variable.

 - ❖ **Error Handling:** Gawk provides error handling mechanisms that allow you to deal with exceptional situations or invalid data gracefully.

-
- ❖ **Formatted Output:** We can use the **printf** function to format the output, allowing you to control the appearance of your results, align columns, and format numeric values.

 - ❖ **Special Patterns:** Gawk includes special patterns like **'BEGIN'** (actions before processing starts), **'END'** (actions after processing ends), and **'NR'** (matching by record number), which are useful for implementing specific behaviors.

 - ❖ **Data Processing:** Gawk is not limited to text processing; you can use it for more advanced data manipulation tasks such as computing statistics, parsing structured data formats, and generating reports.

 - ❖ **Regular Expressions in Awk:** Gawk, as an extension of Awk, supports advanced regular expressions, providing more powerful text pattern matching capabilities.

Using variables in gawk

Assigning Values to Variables: You can assign values to variables in Gawk using the '=' operator. Variables do not require explicit data types; Gawk dynamically determines the type based on the assigned value.

Script: *variable_name = value*

Example

my_number = 42

To assign a string: *my_string = "Hello, World!"*

Using Variables in Your Script: We can use variables within your Awk script by referencing them with a dollar sign ('\$') followed by the variable name. Variables are used to store and manipulate data in your script.

Example

Using a variable in a print statement

print "The value of my_number is: " my_number

Using a variable in an if condition

if (my_number > 10) {

```
print "my_number is greater than 10"
```

```
}
```

Built-in Variables: Gawk provides several built-in variables that are automatically populated and can be used in your script. For example:

- **'NF':** The number of fields in the current record.
- **'NR':** The current record number.
- **'FS':** The input field separator.
- **'OFS':** The output field separator.
- **'RS':** The input record separator.
- **'ORS':** The output record separator.

User-Defined Functions with Variables: We can also create your own user-defined functions that accept and return variables. This allows you to modularize your code and perform specific operations.

Example

```
# User-defined function that doubles a number
function double(x) {
  return x * 2
}
# Use the function with a variable
my_number = 7
result = double(my_number)
print "Double of " my_number " is " result
```

Using structured commands

Many programs require some sort of logic flow control between the commands in the script. There are many commands that allows the script to skip over executed commands based on tested conditions. this commands called as structured commands.

Conditionals (if-else): Conditional statements allow you to execute different actions based on whether a certain condition is met.

Example

```
if (num % 2 == 0) {
  print "Even"
} else {
```

```
print "Odd"
```

```
}
```

Loops (while and for)

Loops allow you to repeat a set of commands multiple times. Gawk provides both 'while' and 'for' loops.

Example for: While Loop

```
i = 1
while (i <= 5) {
print i
i++
}
```

Example for: for Loop

```
for (i = 1; i <= 5; i++) {
print i
}
```

User-Defined Functions: If we define our own functions in Gawk. Functions allow you to encapsulate a block of code and execute it by calling the function.

Example, a function to calculate the square of a number

```
function square(x) {
return x * x
}
num = 4
result = square(num)
print "The square of " num " is " result
```

Control Flow Commands (break and continue)

Gawk supports the 'break' and 'continue' statements for controlling the flow within loops. 'break' is used to exit a loop prematurely, and 'continue' is used to skip the current iteration and move to the next one.

Example

```
for (i = 1; i <= 10; i++) {  
    if (i % 2 == 0) {  
        continue  
    }  
    print i  
}
```

Formatting the printing

Using echo with Escape Sequences: We can use escape sequences with the **'echo'** command to format the output. Common escape sequences include:

- **'\n'**: Newline
- **'\t'**: Tab
- **'\b'**: Backspace
- **'\r'**: Carriage return

Example

```
echo "First Line\nSecond Line"
```

Output

First Line

Second Line

Using printf: The **'printf'** command provides extensive control over output formatting. To specify the format of the output using format specifiers. For example, **'%s'** is used for strings, **'%d'** for integers, and **'%f'** for floating-point numbers.

Example

```
printf "Name: %s, Age: %d\n" "John" 30
```

Output

Name: John, Age: 30

Controlling Field Width and Alignment: The width and alignment of fields in **'printf'** to make output columns align neatly. To specify a field width, use a number between

the ‘%’ sign and the format specifier. To use ‘-’ to specify left alignment, or omit it for right alignment.

Example

```
printf "%-10s %5s\n" "Name" "Age"  
printf "%-10s %5d\n" "John" 30  
printf "%-10s %5d\n" "Alice" 25
```

Output

```
Name   Age  
John   30  
Alice  25
```

Colorizing Output: We can add color to your output using ANSI escape codes. For example, to print text in red.

Example

```
echo -e "\e[31mThis is red text\e[0m"
```

Output

This will display "This is red text" in red color.

Formatting Variables with printf: If we format variables using ‘printf’ as well. This is especially useful for aligning columns when printing tabular data.

Example

```
name="John"  
age=30  
printf "%-10s %5d\n" "$name" "$age"
```

Alignment with column: If we want to align columns in a more structured way, you can use the ‘column’ command. It can automatically format and align columns from input text.

Example

```
echo -e "Name\tAge\nJohn\t30\nAlice\t25" | column -t -s $'\t'
```

Output

```
Name Age  
John 30  
Alice 25
```

Working with functions

The function is a command in Linux that is used to create functions or methods. It is used to perform a specific task or a set of instructions. It allows users to create

shortcuts for lengthy tasks making the command-line experience more efficient and convenient.

Defining Functions

Example

```
say_hello() {  
    echo "Hello, World!"  
}
```

Calling Functions: To call a function, simply use its name followed by parentheses

Example for calling Function

```
say_hello
```

Passing Arguments to Functions: Functions can accept arguments, which are accessed using the special variables '**\$1**', '**\$2**', and so on, where '**\$1**' represents the first argument, '**\$2**' represents the second argument, and so on. Inside the function, you can refer to these variables.

Example

```
print_arguments() {  
    echo "First argument: $1"  
    echo "Second argument: $2"  
}
```

Output

```
print_arguments "Apple" "Banana"
```

Returning Values from Functions: Functions can return values using the '**return**' statement. The return value is stored in the special variable '**\$?**'.

Example

```
add() {  
    local result=$(( $1 + $2 ))  
    return $result  
}
```

Output

```
add 5 7  
result=$?  
echo "The sum is $result"
```

Local Variables: To avoid variable naming conflicts between functions and the main script, you can declare variables as **'local'** within a function. These local variables are only accessible within the function.

Example

```
calculate() {  
    local result=$(( $1 * 2 ))  
    echo "Inside function: result is $result"  
}  
  
result=10  
calculate $result  
echo "Outside function: result is $result"
```

Using Functions in Your Scripts: To place function definitions anywhere in your script, typically at the top or bottom. Functions can be called from any part of the script.

Example

```
#!/bin/bash  
  
say_hello() {  
    echo "Hello, World!"  
}  
  
print_arguments() {  
    echo "First argument: $1"  
    echo "Second argument: $2"  
}  
  
add() {  
    local result=$(( $1 + $2 ))  
    return $result  
}  
  
say_hello  
print_arguments "Apple" "Banana"  
  
add 5 7  
result=$?  
echo "The sum is $result"
```

Let us sum up

grep: Searches for patterns in files using regular expressions.

Usage: grep [options] 'pattern' [file]

Options:

-e : Specifies the pattern.

-i : Ignore case (case-insensitive search).

-r or -R : Recursively search directories.

-v : Invert match (show lines that do not match).

egrep: Extended grep that supports extended regular expressions.

Usage: egrep [options] 'pattern' [file]

Note: egrep is now deprecated; use grep -E instead.

grep -E: grep with extended regular expression support.

Usage: grep -E [options] 'pattern' [file]

Options: Same as grep, with extended regex features.

sed: Stream editor for filtering and transforming text.

Check your progress

1. What is a regular expression?*

A) A mathematical formula

B) A sequence of characters that define a search pattern

C) A programming language

D) A file format

2. Which symbol is used to match any single character in a regular expression?*

A) *

B) .

C) ^

D) \$

3. Which character is used to indicate zero or more occurrences of the previous element in a regular expression?*

- A) + B) * C) ? D) |

4. How do you match the beginning of a line in a regular expression?*

- A) ^ B) \$ C) \b D) \A

5. Which sed command allows processing of multiline input?*

- A) n B) N C) p D) d

6. What is the purpose of the hold space in sed?*

- A) To store the current pattern space for later use
B) To delete the current line
C) To append text to the current line
D) To replace text in the current line

7. How do you negate a command in sed?*

- A) By using ! before the command B) By using ^ before the command
C) By using ~ before the command D) By using # before the command

8. Which command in sed is used to jump to a label?*

- A) b B) t C) g D) h

9. What is the basic syntax for replacing text in sed?*

- A) s/pattern/replacement/ B) r/pattern/replacement/
C) c/pattern/replacement/ D) d/pattern/replacement/

Q10. Can sed commands be included directly in shell scripts?

A) Yes

B) No

11. How do you create a sed script file for complex text processing tasks?*

A) By writing commands in a file and using sed -f filename

B) By writing commands directly in the terminal

C) By creating an alias

D) By using the awk command

12. What is gawk primarily used for?*

A) Text editing

B) Text pattern scanning and processing

C) File compression

D) Network management

13. How do you define a variable in gawk?*

A) variable=value

B) var value

C) let variable = value

D) variable := value

14. Which command in gawk allows for conditional execution?*

A) if

B) for

C) while

D) All of the above

15. Which function in gawk is used to format and print output?*

A) print

B) printf

C) echo

D) format

16. How do you define a function in gawk?*

A) function name { ... }

B) def name { ... }

C) func name { ... }

D) name() { ... }

Here are the answers:

1. B) 2. B) 3. B) 4. A) 5. B) 6. A) 7. A) 8. A) 9. A)
10. A) 11. A) 12. B) 13. A) 14. D) 15. B) 16. A)

Self Assessment Questions :

1. How to validate a phone number in Regular Expressions.
2. Comment on sed editor gawk program.
3. What is Regular Expression with example?
4. Explain in detail about Advanced sed commands with examples.
5. Explain advanced gawk commands with examples.
6. How to remove an HTML tag using sed commands
7. Describe about regular expression in detail.
8. What is reexamining gawk? Write about uses of variable in it.

Open source e-content link

<https://www.geeksforgeeks.org/how-to-use-regular-expressions-regex-on-linux/>

<https://data-flair.training/blogs/regular-expression-in-linux/>

Glossary

Usage: sed [options] 'script' [file]

Options:

-e : Allows multiple commands.

-n : Suppresses automatic printing of pattern space.

awk: A programming language for pattern scanning and processing.

Usage: awk [options] 'pattern { action }' [file]

Options:

-F : Set the input field separator.

perl: A programming language with powerful regular expression capabilities.

Usage: perl -pe 'pattern' [file]

Options:

-e : Allows execution of Perl code from the command line.

-p : Loop over lines and print (similar to sed).

find: Searches for files in a directory hierarchy.

Usage: find [path] [options] [expression]

Options:

-name : Match files by name with a pattern.

-regex : Match files by regex pattern.

Regex Syntax and Concepts

^: Anchors the match at the start of a line.

Usage: ^pattern

\$: Anchors the match at the end of a line.

Usage: pattern\$

.: Matches any single character except a newline.

Usage: a.b (matches acb, a1b, etc.)

*: Matches zero or more of the preceding element.

Usage: a* (matches a, aa, aaa, etc.)

+: Matches one or more of the preceding element.

Usage: a+ (matches a, aa, aaa, etc.)

?: Matches zero or one of the preceding element (makes it optional).

Usage: a? (matches `` or a)

{n}: Matches exactly n occurrences of the preceding element.

Usage: a{3} (matches aaa)

{n,}: Matches n or more occurrences of the preceding element.

Usage: a{2,} (matches aa, aaa, aaaa, etc.)

`{n,m}`: Matches between n and m occurrences of the preceding element.

Usage: `a{2,4}` (matches aa, aaa, aaaa)

`[]`: Defines a character class; matches any one of the enclosed characters.

Usage: `[abc]` (matches a, b, or c)

`[^]`: Defines a negated character class; matches any character not enclosed.

Usage: `[^abc]` (matches any character except a, b, or c)

`|`: Acts as a logical OR between patterns.

Usage: `a|b` (matches a or b)

`()`: Groups patterns together.

Usage: `(abc)+` (matches one or more occurrences of abc)

`\\`: Escapes special characters.

Usage: `\\.` (matches a literal dot)

`\\d`: Matches any digit (in extended regex or Perl).

Usage: `\\d` (matches any digit, equivalent to `[0-9]`)

`\\D`: Matches any non-digit (in extended regex or Perl).

Usage: `\\D` (matches any non-digit)

`\\w`: Matches any word character (alphanumeric and underscore, in extended regex or Perl).

e-books

1. "sed & awk: UNIX Power Tools" by Dale Dougherty and Arnold Robbins
2. "Mastering Regular Expressions" by Jeffrey E. F. Friedl

Unit – V

Objectives:

- Expand your scripting capabilities by learning to use and write scripts for alternative shell environments beyond the default /bin/bash.
- Create simple script utilities that can automate common tasks, improve productivity, and handle repetitive operations efficiently.
- Develop scripts to interact with databases, web services, and email systems to automate data management, web scraping, and communication tasks.

5.1 Working with Alternative Shells:

Understanding the dash shell

What Is the dash Shell?

- The **Debian dash shell** has had an interesting past. It's a direct descendant of the ash shell, **a simple copy of the original Bourne shell** available on Unix systems.
- **Kenneth Almquist** created a small-scale version of the **Bourne shell for Unix systems and called it the Almquist shell, which was then shortened to ash.**
- This original version of the **ash shell was extremely small and fast but without many advanced features**, such as command line editing or history features, making it difficult to use as an interactive shell
- **The NetBSD developers** customized the ash shell by **adding several new features**, making it closer to the Bourne shell.
- The **Debian Linux distribution** created its own version of the ash shell (called Debian ash, or *dash*) for inclusion in its version of Linux. For **the most part, dash copies the features of the NetBSD version of the ash shell, providing the advanced command line editing capabilities.**

The dash Shell Features :

The dash command line parameters

TABLE 23-1 The dash Command Line Parameters

| Parameter | Description |
|-----------|---|
| -a | Exports all variables assigned to the shell |
| -c | Reads commands from a specified command string |
| -e | If not interactive, exits immediately if any untested command fails |
| -f | Displays pathname wildcard characters |
| -n | If not interactive, reads commands but doesn't execute them |
| -u | Writes an error message to <code>STDERR</code> when attempting to expand a variable that is not set |
| -v | Writes input to <code>STDERR</code> as it is read |
| -x | Writes each command to <code>STDERR</code> as it is executed |
| -I | Ignores EOF characters from the input when in interactive mode |
| -i | Forces the shell to operate in interactive mode |
| -m | Turns on job control (enabled by default in interactive mode) |
| -s | Reads commands from <code>STDIN</code> (the default behavior if no file arguments are present) |
| -E | Enables the emacs command line editor |
| -V | Enables the vi command line editor |

- **Positional parameters**

Here are the positional parameter variables available for use in the dash shell:

- `$0`: The name of the shell
- `$n`: The *n*th position parameter
- `$*`: A single value with the contents of all the parameters, separated by the first character in the `IFS` environment variable, or a space if `IFS` isn't defined
- `$@`: Expands to multiple arguments consisting of all the command line parameters
- `$#`: The number of positional parameters
- `$?`: The exit status of the most recent command
- `$-`: The current option flags
- `$$`: The process ID (PID) of the current shell
- `$_`: The process ID (PID) of the most recent background command

User-defined environment variables:

- The dash shell also allows you to set your own environment variables. As with bash, you can define a new environment variable on the command line by using the assignment statement:

```
$ testing=10 ; export testing
$ echo $testing
10
$
```

Without the export command, user-defined environment variables are visible only in the current shell or process.

- **The dash built-in commands**

| Command | Description |
|----------|---|
| alias | Creates an alias string to represent a text string |
| bg | Continues specified job in background mode |
| cd | Switches to the specified directory |
| echo | Displays a text string and environment variables |
| eval | Concatenates all arguments with a space |
| exec | Replaces the shell process with the specified command |
| exit | Terminates the shell process |
| export | Exports the specified environment variable for use in all child shells |
| fg | Continues specified job in foreground mode |
| getopts | Obtains options and arguments from a list of parameters |
| hash | Maintains and retrieves a hash table of recent commands and their locations |
| pwd | Displays the value of the current working directory |
| read | Reads a line from STDIN and assign the value to a variable |
| readonly | Reads a line from STDIN to a variable that can't be changed |
| printf | Displays text and variables using a formatted string |
| set | Lists or sets option flags and environment variables |
| shift | Shifts the positional parameters a specified number of times |
| test | Evaluates an expression and returns 0 if true or 1 if false |
| times | Displays the accumulated user and system times for the shell and all shell processes |
| trap | Parses and executes an action when the shell receives a specified signal |
| type | Interprets the specified name and displays the resolution (alias, built-in, command, keyword) |
| ulimit | Queries or sets limits on processes |
| umask | Sets the value of the default file and directory permissions |
| unalias | Removes the specified alias |

- **Scripting in dash**

Using arithmetic

Three ways to express a mathematical operation in the bash shell script:

- **Using the `expr` command:** `expr operation`
- **Using square brackets:** `$(operation)`
- **Using double parentheses:** `$((operation))`

The dash shell supports the `expr` command and the double parentheses method but doesn't support the square bracket method. This can be a problem if you have lots of mathematical operations that use the square brackets.

The proper format for performing mathematical operations in dash shell scripts is to use the double parentheses method:

```
$ cat test5b
#!/bin/dash
# testing mathematical operations

value1=10
value2=15

value3=$(( $value1 * $value2 ))
echo "The answer is $value3"
$ ./test5b
The answer is 150
$
```

Now the shell can perform the calculation properly.

The test command

However, the test command available in the dash shell doesn't recognize the `==` symbol for text comparisons. Instead, **it only recognizes the `=` symbol**. If you use the `==` symbol in your bash scripts, you need to change the text comparison symbol to just a single equal sign:

```

$ cat test7
#!/bin/dash
# testing the = comparison

test1=abcdef
test2=abcdef

if [ $test1 = $test2 ]
then
    echo "They're the same!"
else
    echo "They're different"
fi
$ ./test7
They're the same!
$

```

- **The function Command**

The dash shell doesn't support the function statement. Instead, in the dash shell you must define a function using the function name with parentheses. If you're writing shell scripts that may be used in the dash environment, always define functions using the function name and not the function() statement:

```

$ cat test10
#!/bin/dash
# testing functions

func1() {
    echo "This is an example of a function"
}

count=1
while [ $count -le 5 ]
do
    func1
    count=$(( $count + 1 ))
done
echo "This is the end of the loop"
func1
echo "This is the end of the script"
$ ./test10
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is an example of a function
This is the end of the loop
This is an example of a function
This is the end of the script
$

```

- **The zsh Shell**

Another popular shell that you may run into is the Z shell (called zsh). The zsh shell is an open source Unix shell developed by **Paul Falstad**.

- The following are some of the features that make the zsh shell unique:
 - Improved shell option handling
 - Shell compatibility modes
 - Loadable modules
- A loadable module is the most advanced feature in shell design. **As you've seen in the bash and dash shells, each shell contains a set of built-in commands that are available without the need for external utility programs.**

The zsh shell provides a core set of built-in commands, plus the capability to add more *command modules*.

Parts of the zsh Shell

Shell options Most shells use command line parameters to define the behavior of the shell. The zsh shell uses a few command line parameters to define the operation of the shell, but mostly it uses *options* to customize the behavior of the shell. You can set shell options either on the command line or within the shell itself using the `set` command.

TABLE 23-3 The zsh Shell Command Line Parameters

| Parameter | Description |
|-----------|---|
| -c | Executes only the specified command and exits |
| -i | Starts as an interactive shell, providing a command line interface prompt |
| -s | Forces the shell to read commands from <code>STDIN</code> |
| -o | Specifies command line options |

Built-in commands

TABLE 23-4 The zsh Core Built-In Commands

| Command | Description |
|-----------|---|
| alias | Defines an alternate name for a command and arguments |
| autoload | Preloads a shell function into memory for quicker access |
| bg | Executes a job in background mode |
| bindkey | Binds keyboard combinations to commands |
| builtin | Executes the specified built-in command instead of an executable file of the same name |
| bye | The same as exit |
| cd | Changes the current working directory |
| chdir | Changes the current working directory |
| command | Executes the specified command as an external file instead of a function or built-in command |
| declare | Sets the data type of a variable (same as typeset) |
| dirs | Displays the contents of the directory stack |
| disable | Temporarily disables the specified hash table elements |
| disown | Removes the specified job from the job table |
| echo | Displays variables and text |
| emulate | Sets zsh to emulate another shell, such as the Bourne, Korn, or C shells |
| enable | Enables the specified hash table elements |
| eval | Executes the specified command and arguments in the current shell |
| fc | Selects a range of commands from the history list |
| fg | Executes the specified job in foreground mode |
| float | Sets the specified variable for use as a floating point variable |
| functions | Sets the specified name as a function |
| getln | Reads the next value in the buffer stack and places it in the specified variable |
| getopts | Retrieves the next valid option in the command line arguments and places it in the specified variable |
| hash | Directly modifies the contents of the command hash table |
| history | Lists the commands contained in the history file |
| integer | Sets the specified variable for use as an integer value |
| jobs | Lists information about the specified job or all jobs assigned to the shell process |
| kill | Sends a signal (Default SIGTERM) to the specified process or job |
| let | Evaluates a mathematical operation and assigns the result to a variable |
| limit | Sets or displays resource limits |
| local | Sets the data features for the specified variable |
| log | Displays all users currently logged in who are affected by the watch parameter |
| logout | Same as exit, but works only when the shell is a login shell |
| popd | Removes the next entry from the directory stack |
| exec | Executes the specified command and arguments replacing the current shell process |
| exit | Exits the shell with the specified exit status. If none specified, uses the exit status of the last command |
| export | Allows the specified environment variable names and values to be used in child shell processes |
| false | Returns an exit status of 1 |

| <code>print</code> | Displays variables and text |
|-------------------------|--|
| <code>printf</code> | Displays variables and text using C-style format strings |
| <code>pushd</code> | Changes the current working directory and puts the previous directory in the directory stack |
| <code>pushln</code> | Places the specified arguments into the editing buffer stack |
| <code>pwd</code> | Displays the full pathname of the current working directory |
| <code>read</code> | Reads a line and assigns data fields to the specified variables using the <code>IFS</code> characters |
| <code>readonly</code> | Assigns a value to a variable that can't be changed |
| <code>rehash</code> | Rebuilds the command hash table |
| <code>set</code> | Sets options or positional parameters for the shell |
| <code>setopt</code> | Sets the options for a shell |
| <code>shift</code> | Reads and deletes the first positional parameter and shifts the remaining ones down one position |
| <code>where</code> | Displays the pathname of the specified command if found by the shell |
| <code>Which</code> | Displays the pathname of the specified command using csh-style output |
| <code>zcompile</code> | Compiles the specified function or script for faster autoloading |
| Command | Description |
| <code>source</code> | Finds the specified file and copies its contents into the current location |
| <code>suspend</code> | Suspends the execution of the shell until it receives a <code>SIGCONT</code> signal |
| <code>test</code> | Returns an exit status of 0 if the specified condition is <code>TRUE</code> |
| <code>times</code> | Displays the cumulative user and system times for the shell and processes that run in the shell |
| <code>trap</code> | Blocks the specified signals from being processed by the shell and executes the specified commands if the signals are received |
| <code>true</code> | Returns a zero exit status |
| <code>ttyctl</code> | Locks and unlocks the display |
| <code>type</code> | Displays how the specified command would be interpreted by the shell |
| <code>typeset</code> | Sets or displays attributes of variables |
| <code>ulimit</code> | Sets or displays resource limits of the shell or processes running in the shell |
| <code>umask</code> | Sets or displays the default permissions for creating files and directories |
| <code>unalias</code> | Removes the specified command alias |
| <code>unfunction</code> | Removes the specified defined function |
| <code>unhash</code> | Removes the specified command from the hash table |
| <code>unlimit</code> | Removes the specified resource limit |
| <code>unset</code> | Removes the specified variable attribute. |
| <code>unsetopt</code> | Removes the specified shell option |
| <code>wait</code> | Waits for the specified job or process to complete |
| <code>whence</code> | Displays how the specified command would be interpreted by the shell |

- **Add-in modules**

- There's a long list of modules that provide additional built-in commands for the zsh shell, and the list continues to grow as resourceful programmers create new modules.

TABLE 23-5 The zsh Modules

| Module | Description |
|----------------|---|
| zsh/datetime | Additional date and time commands and variables |
| zsh/files | Commands for basic file handling |
| zsh/mapfile | Access to external files via associative arrays |
| zsh/mathfunc | Additional scientific functions |
| zsh/pcre | The extended regular expression library |
| zsh/net/socket | Unix domain socket support |
| zsh/stat | Access to the stat system call to provide system statistics |
| zsh/system | Interface for various low-level system features |
| zsh/net/tcp | Access to TCP sockets |
| zsh/zftp | A specialized FTP client command |
| zsh/zselect | Blocks and returns when file descriptors are ready |
| zsh/zutil | Various shell utilities |

- **Viewing and adding modules**

The `zmodload` command is the interface to the zsh modules. You use this command to view, add, and remove modules from the zsh shell session. Using the `zmodload` command without any command line parameters displays the currently installed modules in your zsh shell:

- `% zmodload`
`zsh/zutil`
`zsh/complete`
`zsh/main`
`zsh/terminfo`
`zsh/zle`
`zsh/parameter`
`%`

Different zsh shell implementations include different modules by default. To add a new module, just specify the module name on the zmodload command line:

```
% zmodload zsh/zftp
```

```
%
```

Scripting with zsh

- **Mathematical operations**

- As you would expect, the zsh shell allows you to perform mathematical functions with ease. In the past, the Korn shell has led the way in supporting mathematical operations by providing support for floating-point numbers. The zsh shell has full support for floating point numbers in all its mathematical operations! Performing calculations The zsh shell supports two methods for performing mathematical operations:

- The let command

- Double parentheses

- When you use the let command, you **should enclose the operation in double quotation marks** to allow for spaces:

```
% let value1=" 4 * 5.1 / 3.2 "
```

```
% echo $value1
```

```
6.3750000000
```

```
%
```

- The second method is to use the double parentheses. This method incorporates two techniques for defining the mathematical operation:

```
% value1=$(( 4 * 5.1 ))
```

```
% (( value2 = 4 * 5.1 ))
```

```
% printf "%6.3f\n" $value1 $value2
```

```
20.400
```

```
20.400
```

```
%
```

Notice that you can place the double parentheses either around just the operation (preceded by a dollar sign) or around the entire assignment statement. Both methods produce the same results.

Mathematical functions

With the zsh shell, built-in mathematical functions are either feast or famine. The default zsh shell doesn't include any special mathematical function. However, **if you install the zsh/mathfunc module, you have more math functions than you'll most likely ever need:**

- ```
% value1=$((sqrt(9)))
zsh: unknown function: sqrt
% zmodload zsh/mathfunc
% value1=$((sqrt(9)))
% echo $value1
3.
%
```

- **Structured commands**

- The zsh shell provides the usual set of structured commands for your shell scripts:
  - if-then-else statements
  - for loops (including the C-style)
  - while loops
  - until loops
  - select statements
  - case statements

The zsh shell uses the same syntax for each of these structured commands that you're used to from the bash shell. The zsh shell also includes a different structured command called `repeat`. The `repeat` command uses this format:  
`repeat param do commands done`

```
% cat test1
#!/bin/zsh
using the repeat command

value1=$((10 / 2))
repeat $value1
do
 echo "This is a test"
done
$./test1
This is a test
This is a test
This is a test
This is a test
This is a test
%
```

## Functions

The zsh shell supports the creation of your own functions either using the function command or by defining the function name with parentheses:

```
% function functest1 {
> echo "This is the test1 function"
}
% functest2() {
> echo "This is the test2 function"
}
% functest1
This is the test1 function
% functest2
This is the test2 function
%
```

## 5.2 Writing Simple Script Utilities:

### Automating backups

- Whether you're responsible for a Linux system in a business environment or just using it at home, the loss of data can be catastrophic.
- To help prevent bad things from happening, it's always a good idea to perform regular backups (or archives).

- 
- However, what's a good idea and what's practical are often two separate things.
  - Trying to arrange a backup schedule to store important files can be a challenge.

### **Archiving data files**

- If you're using your Linux system to work on an important project, you can create a shell script that automatically takes snapshots of specific directories.
- Designating these directories in a configuration file allows you to change them when a particular project changes.
- This helps avoid a time consuming restore process from your main archive files.
- This section shows you how to create an automated shell script that can take snapshots of specified directories and keep an archive of your data's past versions.

### **Obtaining the required functions:**

- The workhorse for archiving data in the Linux world is the tar command .
- The tar command is used to archive entire directories into a single file. Here's an example of creating an archive file of a working directory using the tar command:
- The tar command responds with a warning message that it's removing the leading forward slash from the pathname to convert it from an absolute pathname to a relative pathname
- This allows you to extract the tar archived files anywhere you want in your filesystem.
- You can accomplish this by redirecting STDERR to the /dev/null file
- Because a tar archive file can consume lots of disk space, it's a good idea to compress the file. You can do this by simply adding the -z option. This compresses the tar archive file into a gzipped tar file, which is called a tarball.

---

## Creating a daily archive location

- If you are just backing up a few files, it's fine to keep the archive in your personal directory.
- However, if several directories are being backed up, it is best to create a central repository archive directory

```
$ sudo mkdir /archive
[sudo] password for Christine:
$
$ ls -ld /archive
drwxr-xr-x. 2 root root 4096 Aug 27 14:10 /archive
$
```

- After you have your central repository archive directory created, you need to grant access to it for certain users. If you do not do this, trying to create files in this directory fails, as shown here:

```
$ mv Files_To_Backup /archive/
mv: cannot move 'Files_To_Backup' to
'/archive/Files_To_Backup': Permission denied
$
```

- You could grant the users needing to create files in this directory permission via sudo or create a user group. In this case, a special user group is created, Archivers:

```
$ sudo groupadd Archivers
$
$ sudo chgrp Archivers /archive
$
$ ls -ld /archive
```



```
drwxr-xr-x. 2 root Archivars 4096 Aug 27 14:10 /archive
$
$ sudo usermod -aG Archivars Christine
[sudo] password for Christine:
$
$ sudo chmod 775 /archive
$
$ ls -ld /archive
drwxrwxr-x. 2 root Archivars 4096 Aug 27 14:10 /archive
$
```

After a user has been added to the Archivars group, the user must log out and log back in for the group membership to take effect. Now files can be created by this group's members without the use of super-user privileges:

```
$ mv Files_To_Backup /archive/
$
$ ls /archive
Files_To_Backup
$
```

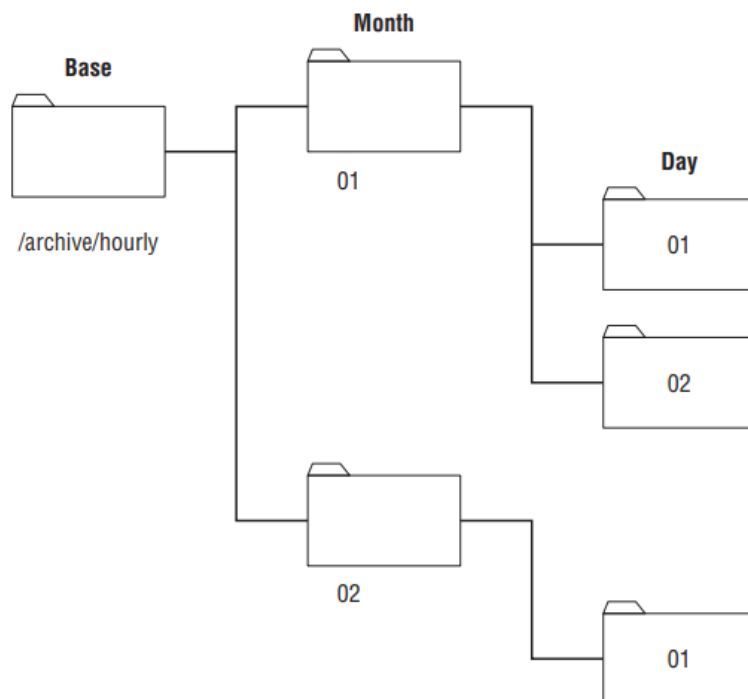
## Creating an hourly archive script

- If you are in a high-volume production environment where files are changing rapidly, a daily archive might not be good enough.
- If you want to increase the archiving frequency to hourly, you need to take another item into consideration

When backing up files hourly and trying to use the date command to timestamp each tarball, things can get pretty ugly pretty quickly. Sifting through a directory of tarballs with filenames looking like this is tedious:

```
archive010211110233.tar.gz
```

Instead of placing all the archive files in the same folder, you can create a directory hierarchy for your archived files.



- First, the new directory /archive/hourly must be created, along with the appropriate permissions set upon it.

```

$ sudo mkdir /archive/hourly
[sudo] password for Christine:
$

```

```

$ sudo chgrp Archivers /archive/hourly
$
$ ls -ld /archive/hourly/
drwxr-xr-x. 2 root Archivers 4096 Sep 2 09:24 /archive/hourly/
$
$ sudo chmod 775 /archive/hourly
$
$ ls -ld /archive/hourly
drwxrwxr-x. 2 root Archivers 4096 Sep 2 09:24 /archive/hourly
$

```

]

After the new directory is set up, the `Files_To_Backup` configuration file for the hourly archives can be moved to the new directory:

```
$ cat Files_To_Backup
/usr/local/Production/Machine_Errors
/home/Development/Simulation_Logs
$
$ mv Files_To_Backup /archive/hourly/
$
```

- **Running the hourly archive script**

- As with the `Daily_Archive.sh` script, it's a good idea to test the `Hourly_Archive.sh` script before putting it in the cron table. Before the script is run, the permissions must be modified. Also, the hour and minute is checked via the `date` command. Having the current hour and minute allows the final archive filename to be verified for correctness:

```
$ chmod u+x Hourly_Archive.sh
$
$ date +%k%M
1011
$
$./Hourly_Archive.sh
```

Starting archive...

Archive completed

Resulting archive file is: `/archive/hourly/09/02/archive1011.tar.gz`

```
$
$ ls /archive/hourly/09/02/
archive1011.tar.gz
$
```

## Managing User Accounts:

Managing user accounts is much more than just adding, modifying, and deleting accounts.

- You must also consider security issues, the need to preserve work, and the accurate management of the accounts.
- This can be a time-consuming task.

---

## Obtaining the required functions:

Deleting an account is the more complicated accounts management task. When deleting an account, at least four separate actions are required:

1. Obtain the correct user account name to delete.
2. Kill any processes currently running on the system that belongs to that account.
3. Determine all files on the system belonging to the account.
4. Remove the user account.

## Obtaining the required functions

Deleting an account is the more complicated accounts management task. When deleting an account, at least four separate actions are required:

1. Obtain the correct user account name to delete.
  2. Kill any processes currently running on the system that belongs to that account.
  3. Determine all files on the system belonging to the account.
  4. Remove the user account.
- It's easy to miss a step. The shell script utility in this section helps you avoid making such mistakes.

## Getting the correct account name :

The first step in the account deletion process is the most important: **obtaining the correct user account name to delete**. Because this is an interactive script, you can use the read command to obtain the account name. you can use the **-t option on the read command and timeout after giving the script user 60 seconds** to answer the question:

- `echo "Please enter the username of the user "`  
`echo -e "account you wish to delete from system: \c"`  
`read -t 60 ANSWER`

---

## Creating a function to get the correct account name:

The first thing you need to do is declare the function's name, `get_answer`. Next, clear out any previous answers to questions your script user gave using the `unset` command

```
function get_answer {
```

```
#
```

```
unset ANSWER
```

- To ask the script user what account to delete, a few variables must be set and the `get_answer` function should be called. Using the new function makes the script code much simpler:
- ```
LINE1="Please enter the username of the user "  
LINE2="account you wish to delete from system:"  
get_answer  
USER_ACCOUNT=$ANSWER
```

Verifying the entered account name

Because of potential typographical errors, the user account name that was entered should be verified. This is easy because the code is already in place to handle asking a question:

```
LINE1="Is $USER_ACCOUNT the user account "  
LINE2="you wish to delete from the system? [y/n]"  
get_answer
```

```
case $ANSWER in  
y|Y|YES|yes|Yes|yEs|yeS|YEs|yES )  
#  
;;  
*)  
    echo  
    echo "Because the account, $USER_ACCOUNT, is not "  
    echo "the one you wish to delete, we are leaving the script..."  
    echo  
    exit  
;;  
esac
```

Determining whether the account exists

- The user has given us the name of the account to delete and has verified it.
- Now is a good time to double-check that the user account really exists on the system.
- The `-w` option allows an exact word match for this particular user account:
`USER_ACCOUNT_RECORD=$(cat /etc/passwd | grep -w $USER_ACCOUNT)`

Removing any account processes

- The script has obtained and verified the correct name of the user account to be deleted.
- In order to remove the user account from the system, the account cannot own any processes currently running.
- Thus, the next step is to find and kill off those processes.
- Here the script can use the `ps` command and the `-u` option to locate any running processes owned by the account.
 - `ps -u $USER_ACCOUNT >/dev/null #Are user processes running?`

Finding account files :

- When a user account is deleted from the system, it is a good practice to archive all the files that belonged to that account.
- Along with that practice, it is also important to remove the files or assign their ownership to another account.

Removing the account

Finally, we get to the main purpose of our script, actually removing the user account from the system. Here the `userdel` command is used:

```
userdel $USER_ACCOUNT
```

Monitoring Disk Space:

- One of the biggest problems with multi-user Linux systems is the amount of available disk space.

-
- In some situations, such as in a file-sharing server, disk space can fill up almost immediately just because of one careless user.

Obtaining the required functions:

- The first tool you need to use is the **du** command. This command displays the **disk usage for individual files and directories**.

The **-s** option lets you summarize totals at the directory level. This comes in handy when calculating the total disk space used by an individual user. Here's what it looks like to use the **du** command to summarize each user's **\$HOME** directory for the **/home** directory contents:

```
$ sudo du -s /home/*
[sudo] password for Christine:

4204    /home/Christine
56      /home/Consultant
52      /home/Development
4       /home/NoSuchUser
96      /home/Samantha
36      /home/Timothy
1024    /home/user1
$
```

The **-s** option works well for users' **\$HOME** directories, but what if we wanted to view disk consumption in a system directory such as **/var/log**?

```
$ sudo du -s /var/log/*
4       /var/log/anaconda.ifcfg.log
20      /var/log/anaconda.log
32      /var/log/anaconda.program.log
108     /var/log/anaconda.storage.log
40      /var/log/anaconda.syslog
56      /var/log/anaconda.xlog
116     /var/log/anaconda.yum.log
4392    /var/log/audit
4       /var/log/boot.log
[...]
$
```

-
- The listing quickly becomes too detailed. **The -S (capital S) option works better for our purposes here, providing a total for each directory and subdirectory individually.** This allows you to pinpoint problem areas quickly:

```
$ sudo du -S /var/log/
4      /var/log/ppp
4      /var/log/sss
3020   /var/log/sa
80     /var/log/prelink
4      /var/log/samba/old
4      /var/log/samba
4      /var/log/ntpstats
4      /var/log/cups
4392   /var/log/audit
420    /var/log/gdm
4      /var/log/httpd
152    /var/log/ConsoleKit
2976   /var/log/
$
```

Because we are interested in the directories consuming the biggest chunks of disk space, the sort command is used on the listing produced by du: **The -n option allows you to sort numerically. The -r option lists the largest numbers first (reverse order).** This is perfect for finding the largest disk consumers.

```
$ sudo du -S /var/log/ | sort -rn
4392   /var/log/audit
```

```
3020   /var/log/sa
2976   /var/log/
420    /var/log/gdm
152    /var/log/ConsoleKit
80     /var/log/prelink
4      /var/log/sss
4      /var/log/samba/old
4      /var/log/samba
4      /var/log/ppp
4      /var/log/ntpstats
4      /var/log/httpd
4      /var/log/cups
$
```

Creating the script:

- To save time and effort, the script creates a report for multiple designated directories.
- A variable to accomplish this called CHECK_DIRECTORIES is used. For our purposes here, the variable is set to just two directories:

```
CHECK_DIRECTORIES="/var/log /home"
```

- Each time the for loop iterates through the list of values in the variable CHECK_DIRECTORIES, it assigns to the DIR_CHECK variable the next value in the list:

```
for DIR_CHECK in $CHECK_DIRECTORIES
do
[... ]
    du -S $DIR_CHECK
[... ]
done
```

Running the script:

```
$ ls -l Big_Users.sh
-rw-r--r--. 1 Christine Christine 910 Sep  3 08:43 Big_Users.sh
$
$ sudo bash Big_Users.sh
[sudo] password for Christine:
$
$ ls disk_space*.rpt
disk_space_090314.rpt
$
$ cat disk_space_090314.rpt
Top Ten Disk Space Usage
```

— for /var/log /home Directories

The /var/log Directory:

```
1:      4496    /var/log/audit
2:      3056    /var/log
3:      3032    /var/log/sa
4:       480    /var/log/gdm
5:      152     /var/log/ConsoleKit
6:       80     /var/log/prelink
7:        4     /var/log/sss
8:        4     /var/log/samba/old
9:        4     /var/log/samba
10:     4      /var/log/ppp
```

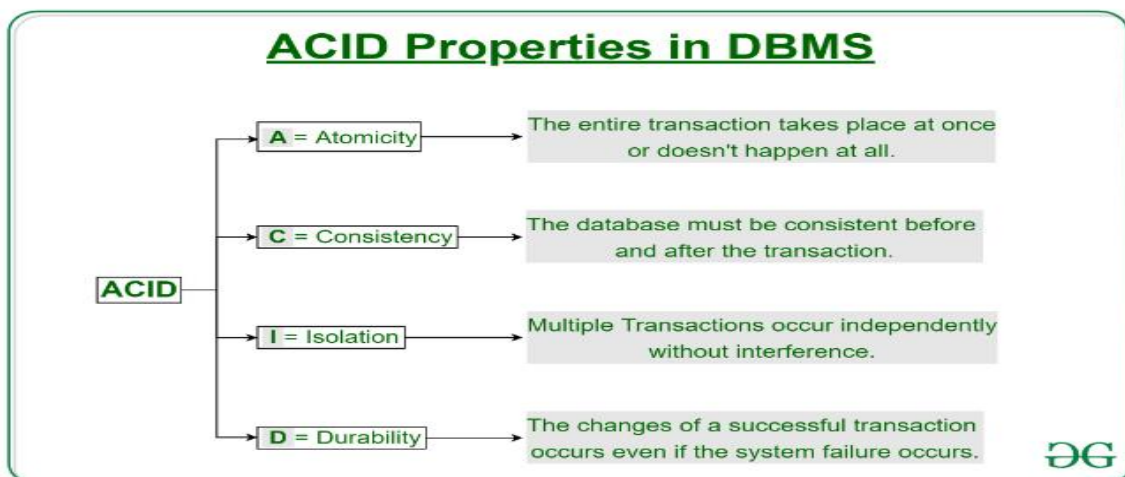
The /home Directory:

```
1:      34084   /home/Christine/Documents/temp/reports/archive
2:      14372   /home/Christine/Documents/temp/reports
3:      4440    /home/Timothy/Project__42/log/universe
4:      4440    /home/Timothy/Project_254/Old_Data/revision.56
5:      4440    /home/Christine/Documents/temp/reports/report.txt
6:      3012    /home/Timothy/Project__42/log
7:      3012    /home/Timothy/Project_254/Old_Data/data2039432
8:      2968    /home/Timothy/Project__42/log/answer
9:      2968    /home/Timothy/Project_254/Old_Data/data2039432/answer
10:     2968    /home/Christine/Documents/temp/reports/answer
$
```

5.3 Producing Scripts for Database, Web, and E-Mail

Data base:

- A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS).



Atomicity:

- By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort:** If a transaction aborts, changes made to the database are not visible.

—**Commit:** If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing'.

| | |
|---|---|
| Before: X : 500 | Y: 200 |
| Transaction T | |
| T1 | T2 |
| Read (X) $X := X - 100$ Write (X) | Read (Y) $Y := Y + 100$ Write (Y) |
| After: X : 400 | Y : 300 |

Consistency:

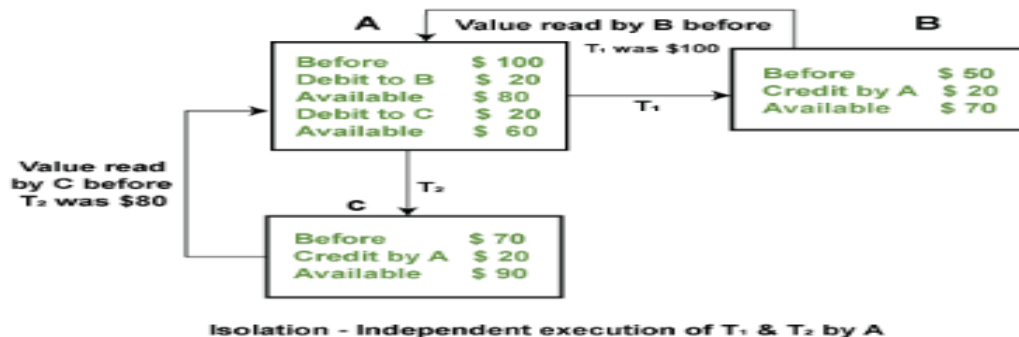
- This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
The total amount before and after the transaction must be maintained.

Total **before T** occurs = $500 + 200 = 700$.

Total **after T** occurs = $400 + 300 = 700$.

Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

Isolation: The term 'isolation' means separation. In DBMS, Isolation is the property of a database where no data should affect the other one and may occur concurrently. In short, the operation on one database should begin when the operation on the first database gets complete. It means if two operations are being performed on two different databases, they may not affect the value of one another. In the case of transactions, when two or more transactions occur simultaneously, the consistency should remain maintained. Any changes that occur in any particular transaction will not be seen by other transactions until the change is not committed in the memory.



Durability: Durability ensures the permanency of something. In DBMS, the term durability ensures that the data after the successful execution of the operation becomes permanent in the database. The durability of the data should be so perfect that even if the system fails or leads to a crash, the database still survives.

- **Using a MySQL Database:**

- system@system-virtual-machine:~/Riyaz\$ sudo apt update

- system@system-virtual-machine:~/Riyaz\$ sudo apt install postgresql

postgresql-contrib

system@system-virtual-machine:~/Riyaz\$ sudo -i -u postgres

sudo] password for system:

postgres@system-virtual-machine:~\$ psql

psql (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))

Type "help" for help.

The mysql commands:

The mysql program uses two different types of commands:

- **Special mysql commands**
- **Standard SQL statements**

The mysql program uses its own set of commands that let you easily control the environment and retrieve information about the MySQL server. The mysql commands use either a full name (such as status) or a shortcut (such as \s).

```
mysql> \s
```

```
-----
```

```
mysql Ver 14.14 Distrib 5.5.38, for debian-linux-gnu (i686) using readline 6.3
```

```
Connection id:
```

```
Current database:
```

```
Current user: root@localhost
```

```
SSL: Not in use
```

```
Current pager: stdout
```

```
Using outfile: "
```

```
Using delimiter: ;
```

```
Server version: 5.5.38-0ubuntu0.14.04.1 (Ubuntu)
```

```
Protocol version: 10
```

```
Connection: Localhost via UNIX socket
```

```
Server characterset: latin1
```

```
Db characterset: latin1
```

```
Client characterset: utf8
```

```
Conn. characterset: utf8
```

```
UNIX socket: /var/run/mysqld/mysqld.sock
```

```
Uptime: 2 min 24 sec
```

```
Threads: 1 Questions: 575 Slow queries: 0 Opens: 421 Flush tables: 1
```

```
Open tables: 41 Queries per second avg: 3.993
```

```
mysql> SHOW DATABASES;
+-----+
| Database          |
+-----+
| information_schema |
| mysql             |
+-----+
2 rows in set (0.04 sec)
```

```
mysql> USE mysql;
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_mysql  |
+-----+
| columns_priv     |
| db               |
| func             |
| help_category    |
| help_keyword     |
| help_relation    |
```

```
| host             |
| proc            |
| procs_priv      |
| tables_priv     |
| time_zone       |
| time_zone_leap_second |
| time_zone_name  |
| time_zone_transition |
| time_zone_transition_type |
| user            |
+-----+
17 rows in set (0.00 sec)
mysql>
```

- **Creating a database:**

The MySQL server organizes data into *databases*. A database usually holds the data for a single application, separating it from other applications that use the database server.

- Creating a separate database for each shell script application helps eliminate confusion and data mix-ups.

Here's the SQL statement required to create a new database:

```
CREATE DATABASE name;
```

- `mysql> CREATE DATABASE mytest;`
Query OK, 1 row affected (0.02 sec)

```
mysql> SHOW DATABASES;
+-----+
| Database           |
+-----+
| information_schema |
| mysql              |
| mytest             |
+-----+
3 rows in set (0.01 sec)
```

Creating a table

The MySQL server is considered a *relational* database. In a relational database, data is organized by *data fields*, *records*, and *tables*. A data field is a single piece of information, such as an employee's last name or a salary. A record is a collection of related data fields, such as the employee ID number, last name, first name, address, and salary. Each record indicates one set of the data fields.

```
mysql> CREATE TABLE employees (
  -> empid int not null,
  -> lastname varchar(30),
  -> firstname varchar(30),
  -> salary float,
  -> primary key (empid));
Query OK, 0 rows affected (0.14 sec)
```

TABLE 25-1 MySQL Data Types

| Data Type | Description |
|-----------|--|
| char | A fixed-length string value |
| varchar | A variable-length string value |
| int | An integer value |
| float | A floating-point value |
| boolean | A Boolean true/false value |
| date | A date value in YYYY-MM-DD format |
| time | A time value in HH:mm:ss format |
| timestamp | A date and time value together |
| text | A long string value |
| BLOB | A large binary value, such as an image or video clip |

- The empid data field also specifies a *data constraint*. A data constraint restricts what type of data you can enter to create a valid record. The not null data constraint indicates that every record must have an empid value specified. Finally, the primary key defines a data field that uniquely identifies each individual record. This means that each data record must have a unique empid value in the table.
- **Inserting and deleting data**
- you use the INSERT SQL command to insert new data records into the table. Each INSERT command must specify the data field values for the MySQL

-
- server to accept the record. Here's the format of the **INSERT SQL** command:
INSERT INTO table VALUES (...)

 - **mysql> INSERT INTO employees VALUES (1, 'Blum', 'Rich', 25000.00);**
Query OK, 1 row affected (0.35 sec)

mysql> INSERT INTO employees VALUES (2, 'Blum', 'Barbara', 45000.00);
Query OK, 1 row affected (0.00 sec)
You should now have two data records in your table

 - Here's the basic **DELETE** command format:
DELETE FROM table;
To just specify a single record or a group of records to delete, you must use the **WHERE** clause. The **WHERE** clause allows you to create a filter that identifies which records to remove. You use the **WHERE** clause like this:

 - **DELETE FROM employees WHERE empid = 2;**
This restricts the deletion process to all the records that have an **empid** value of 2. When you execute this command, the **mysql** program returns a message indicating how many records matched the filter:

 - **mysql> DELETE FROM employees WHERE empid = 2;**
Query OK, 1 row affected (0.29 sec)
As expected, only one record matched the filter and was removed.
Querying data

 - Here's the basic format of a **SELECT** statement:
SELECT datafields FROM table

```
mysql> SELECT * FROM employees;
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
|      1 | Blum     | Rich      | 25000  |
|      2 | Blum     | Barbara   | 45000  |
|      3 | Blum     | Katie Jane | 34500  |
|      4 | Blum     | Jessica   | 52340  |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

You can use one or more modifiers to define how the database server returns the data requested by the query. Here's a list of commonly used modifiers:

- WHERE: Displays a subset of records that meet a specific condition
 - ORDER BY: Displays records in a specified order
 - LIMIT: Displays only a subset of records
- Sending commands to the server After establishing the connection to the server, you'll want to send commands to interact with your database. There are two methods to do this:
 - Send a single command and exit.
 - Send multiple commands.

To send a single command, you must include the command as part of the mysql command line. For the mysql command, you do this using the -e parameter:

```
$ cat mtest1
#!/bin/bash
# send a command to the MySQL server
MYSQL=$(which mysql)
$MYSQL mytest -u test -e 'select * from employees'
```

```

$ ./mtest1
+-----+-----+-----+-----+
| empid | lastname | firstname | salary |
+-----+-----+-----+-----+
|      1 | Blum     | Rich     | 25000  |
|      2 | Blum     | Barbara  | 45000  |
|      3 | Blum     | Katie Jane | 34500  |
|      4 | Blum     | Jessica  | 52340  |
+-----+-----+-----+-----+
$

```

If you need to send more than one SQL command, you can use file redirection. To redirect lines in the shell script, you must define an *end of file* string. The end of file string indicates the beginning and end of the redirected data.

This is an example of defining an end of file string, with data in it:

```

$ cat mtest2
#!/bin/bash
# sending multiple commands to MySQL

MYSQL=$(which mysql)
$MYSQL mytest -u test <<EOF
show tables;
select * from employees where salary > 40000;
EOF
$ ./mtest2
Tables_in_test
employees
empid  lastname  firstname  salary
2      Blum      Barbara    45000
4      Blum      Jessica    52340
$

```

- **Formatting data**

The standard output from the `mysql` command doesn't lend itself to data retrieval. If you need to actually do something with the data you retrieve, you need to do some fancy data manipulation. This section describes some of the tricks you can use to help extract data from your database reports.

- The `mysql` program also supports an additional popular format, called Extensible Markup Language (XML). This language uses HTML-like tags to

-
- identify data names and values. For the mysql program, you do this using the -X command line parameter:

```
$ mysql mytest -u test -X -e 'select * from employees where empid = 1'
<?xml version="1.0"?>

<resultset statement="select * from employees">
<row>
  <field name="empid">1</field>
  <field name="lastname">Blum</field>
  <field name="firstname">Rich</field>
  <field name="salary">25000</field>
</row>
</resultset>
$
```

Using XML, you can easily identify individual rows of data, along with the individual data values in each record. You can then use standard Linux string handling functions to extract the data you need!

Using the Web:

- Almost as old as the Internet itself, the Lynx program was **created in 1992 by students at the University of Kansas as a text-based browser**. Because it's text-based, the Lynx program allows you to browse websites directly from a terminal session, replacing the fancy graphics on web pages with HTML text tags.
- Lynx uses the standard keyboard keys to navigate around the web page. **Links appear as highlighted text within the web page. Using the right-arrow key allows you to follow a link to the next web page.**

Installing Lynx

- Even though the **Lynx program is somewhat old**, it's still in active development. At the time of this writing, the **latest version of Lynx is version 2.8.8, released in June 2010**, with a new release in development. Because of

-
- its popularity among shell script programmers, many Linux distributions install the Lynx program in their default installations.

- **The lynx command line:**

The lynx command line command is extremely versatile in what information it can retrieve from the remote website. When you view a web page in your browser, you're only seeing part of the information that's transferred to your browser. Web pages consist of three types of data elements:

- HTTP headers
- Cookies
- HTML content

- *HTTP headers* provide information about the type of data sent in the connection, the server sending the data, and the type of security used in the connection. If you're sending special types of data, such as video or audio clips, the server identifies that in the HTTP headers. The Lynx program allows you to view all the HTTP headers sent within a web page session.
- If you've done any type of web browsing, no doubt you're familiar with web page *cookies*. Websites use cookies to store data about your website visit for future use. Each individual site can store information, but it can only access the information it sets. The lynx command provides options for you to view cookies sent by web servers, as well as reject or accept specific cookies sent from servers.
- The Lynx program allows you to view the actual HTML content of the web page in three different formats:
 - In a **text-graphics display on the terminal session** using the curses graphical library
 - As a text file, **dumping the raw data** from the web page
 - As a text file, **dumping the raw HTML source code** from the web page

- **The Lynx configuration file**

- The lynx command reads a configuration file for many of its parameter settings. By default, this file is located at /usr/local/lib/lynx.cfg, although you'll find that many Linux distributions change this to the /etc directory (/etc/lynx.cfg) (the Ubuntu distribution places the lynx.cfg file in the /etc/lynx-cur folder). The lynx.cfg configuration file groups related parameters into sections to make finding parameters easier. Here's the format of an entry in the configuration file:

- *PARAMETER:value*

where *PARAMETER* is the full name of the parameter (often, but not always in uppercase letters) and *value* is the value associated with the parameter. such as the ACCEPT_ALL_COOKIES parameter,

- The most common configuration parameters that you can't set on the command line are for the *proxy servers*. **Some networks (especially corporate networks) use a proxy server as a middleman between the client's browser and the destination website server.**

Instead of sending HTTP requests directly to the remote web server, client browsers must send their requests to the proxy server. The proxy server in turn sends the requests to the remote web server, retrieves the results, and forwards them back to the client browser.

Using E-Mail :The main tool you have available for sending e-mail messages from your shell scripts is the Mailx program. Not only can you use it interactively to read and send messages, but you can also use the command line parameters to specify how to send a message.

The Mailx program sends the text from the echo command as the message body. This provides an easy way for you to send messages from your shell scripts. Here's a quick example:

```
$ cat factmail
#!/bin/bash
# mailing the answer to a factorial

MAIL=$(which mailx)

factorial=1
counter=1

read -p "Enter the number: " value
while [ $counter -le $value ]
do
    factorial=$((factorial * $counter))
    counter=$((counter + 1))
done

echo "The factorial of $value is $factorial" | $MAIL -s "Factorial
answer" $USER
echo "The result has been mailed to you."
```

This script does not assume that the Mailx program is located in the standard location. It uses the which command to determine just where the mail program is. After calculating the result of the factorial function, the shell script uses the mail command to send the message to the user-defined \$USER environment variable, which should be the person executing the script.

```
$ ./factmail
Enter the number: 5
The result has been mailed to you.
$
```

You just need to check your mail to see if the answer arrived:

```
$ mail
"/var/mail/rich": 1 message 1 new
>N 1 Rich Blum Mon Sep 1 10:32 13/586 Factorial answer
?
Return-Path: <rich@rich-Parallels-Virtual-Platform>
X-Original-To: rich@rich-Parallels-Virtual-Platform
Delivered-To: rich@rich-Parallels-Virtual-Platform
Received: by rich-Parallels-Virtual-Platform (Postfix, from userid 1000)
 id B4A2A260081; Mon, 1 Sep 2014 10:32:24 -0500 (EST)
Subject: Factorial answer
To: <rich@rich-Parallels-Virtual-Platform>
X-Mailer: mail (GNU Mailutils 2.1)
Message-Id: <20101209153224.B4A2A260081@rich-Parallels-Virtual-Platform>
Date: Mon, 1 Sep 2014 10:32:24 -0500 (EST)
From: rich@rich-Parallels-Virtual-Platform (Rich Blum)

The factorial of 5 is 120
?
```

5.4 What is Python?

- Python is an object-oriented interpreted language that is designed to be easy to use and to aid **Rapid Application Development**. This is achieved by the use of simplified semantics in the language.
- Python was **created at the end of the 1980s, towards the very end of December 1989**, by the **Dutch developer Guido van Rossum**. The majority of the design of the language aims for clarity and simplicity
- If you are using another Linux distribution or Python 3 is not found for any reason, you can install it like this: On RedHat based distributions: **\$ sudo yum install python36** On Debian based distributions:
- **\$ sudo apt-get install python3.6**
- We can see that we are presented with `>>>` the prompt and this is known as the REPL console. We should emphasize that this is a scripting language and, like bash and Perl, we will normally execute code through the text files that we create. Those text files will normally be expected to have a QZ suffix to their name.
- While working with REPL, we can print the version independently by importing a module.
In Perl, we will use the keyword `use`; in bash we will use the command `source`; and

-
- in Python
we use import:

```
>>>import sys
```

With the module loaded, we can now investigate the object-oriented nature of Python by printing the version:

```
>>> sys.version
```

```
>>> import sys
>>> sys.version
'3.2.3 (default, Mar  1 2013, 11:53:50) \n[GCC 4.6.3]'
>>> _
```

We will navigate to the TZT object within our namespace and call the version method from that object.

Finally, to close the REPL, we will use *Ctrl + D* in Linux or *Ctrl + Z* in Windows.

- **Saying Hello World the Python way:**

The Print function includes the newline and we do not need semicolons at the end of the line. We can

see the edited version of **`$HOME/bin/hello.py`** in the following example:

- We will still need to add the execute permission, but we can run the code as earlier using Chmod. This is shown in the following command but we should be a little used to this now:

```
$ chmod u+x $HOME/bin/hello.py
```

Finally, we can now execute the code to see our greeting. Similarly, you can run the file using the Python interpreter from the command line like this:

- **`$ python3 $HOME/bin/hello.py`**
- Or in some Linux distributions, you can run it like this:

```
$ python36 $HOME/bin/hello.py
```

- **Pythonic arguments:**

We should know by now that we will want to pass command-line arguments to Python and we can do this using the BSHW array. However, we are more like bash; with Python we combine the program name into the array with the other arguments.

Python also uses lowercase instead of uppercase in the object name:

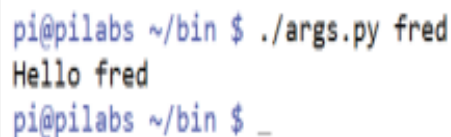
- The `argv` array is a part of the `sys` object
- `sys.argv[0]` is the script name
- `sys.argv[1]` is the first argument supplied to the script
- `sys.argv[2]` is the second supplied argument and so on
- The argument count will always be at least 1, so, keep this in mind when checking for supplied arguments

- **Supplying arguments:**

If we create the `$HOME/bin/args.py` file we can see this in action. The file should be created as follows and made executable:

```
#!/usr/bin/python3
import sys
print("Hello " + sys.argv[1])
```

If we run the script with a supplied argument, we should see something similar to the following screenshot:



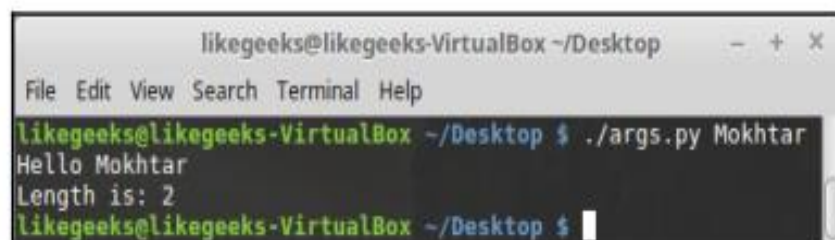
```
pi@pilabs ~/bin $ ./args.py fred
Hello fred
pi@pilabs ~/bin $ _
```

-
- **Counting arguments** : The script name is the first argument at index 0 of the array. So, if we try to count the arguments, then the count should always be at the very least 1.
 - In other words, if we have not supplied arguments, the argument count will be 1. To count the items in an array, we can use the len() function.

If we edit the script to include a new line we will see this work, as follows:

```
#!/usr/bin/python3
import sys
print("Hello " + sys.argv[1])
print( "length is: " + str(len(sys.argv)) )
```

Executing the code as we have earlier, we can see that we have supplied two arguments—the script name and then the string Mokhtar:



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./args.py Mokhtar
Hello Mokhtar
Length is: 2
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Significant whitespace:

The indent level of your code defines the block of code to which it belongs. So far, we have not indented the code we have created past the start of the line. This means that all of the code is at the same indent level and belongs to the same code block.

Rather than using brace brackets or the do and done keywords to define the code block, we use indents. If we indent with two or four spaces or even tabs, then we must stick to those spaces or tabs. When we return to the previous indent level, we return to the previous code block.

```
#!/usr/bin/python3
import sys
count = len(sys.argv)
if ( count > 1 ):
    print("Arguments supplied: " + str(count))
    print("Hello " + sys.argv[1])
print("Exiting " + sys.argv[0])
```

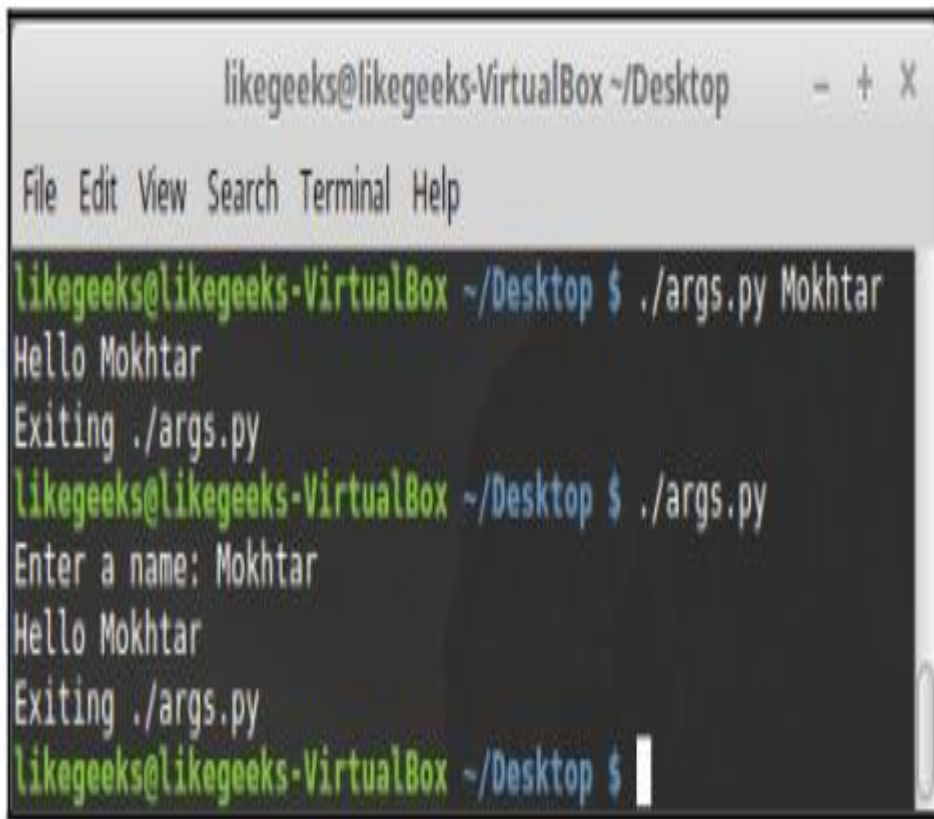
```
pi@pilabs ~/bin $ ./args.py
Exiting ./args.py
pi@pilabs ~/bin $ ./args.py fred
Arguments supplied: 2
Hello fred
Exiting ./args.py
```

Reading user input :

```
#!/usr/bin/python3
import sys
count = len(sys.argv)
name = ''

if ( count == 1 ):
    name = input("Enter a name: ")
else:
    name = sys.argv[1]

print("Hello " + name)
print("Exiting " + sys.argv[0])
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop - + X
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./args.py Mokhtar
Hello Mokhtar
Exiting ./args.py
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./args.py
Enter a name: Mokhtar
Hello Mokhtar
Exiting ./args.py
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Using Python to write to files :

We will start by making a copy of our existing **args.py** We will copy this **\$HOME/bin/file.py**. The new **file.py** should read similar to the following screenshot and have the execute permission set:

```
#!/usr/bin/python3
import sys
count = len(sys.argv)
name = ''

if ( count == 1 ):
    name = input("Enter a name: ")
else:
    name = sys.argv[1]

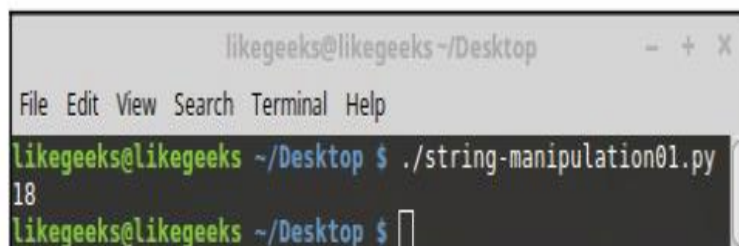
log = open("/tmp/script.log","a")
log.write("Hello " + name + "\n")
log.close()
```

String

manipulation:

Dealing with strings in Python is very simple: you can search, replace, change character case, and perform other manipulations with ease: To search for a string, you can use the find method like this:

```
#!/usr/bin/python3
str = "Welcome to Python scripting world"
print(str.find("scripting"))
```

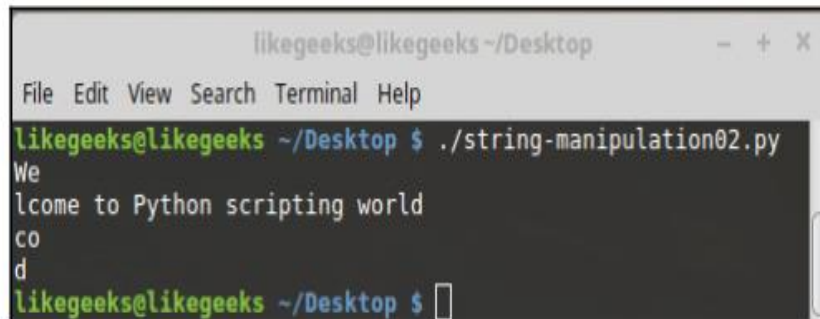


```
likegeeks@likegeeks ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks ~/Desktop $ ./string-manipulation01.py
18
likegeeks@likegeeks ~/Desktop $
```

The string count in Python starts from zero too, so the position of the word `scripting` is at 18.

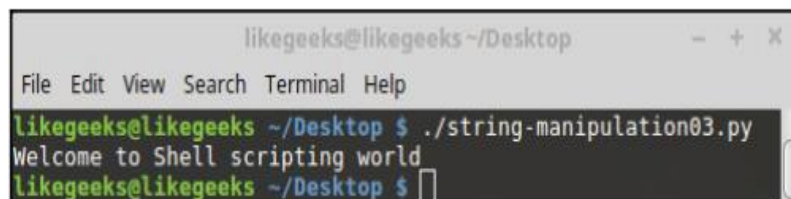
— You can get a specific substring using square brackets like this:

```
#!/usr/bin/python3
str = "Welcome to Python scripting world"
print(str[:2]) # Get the first 2 letters (zero based)
print(str[2:]) # Start from the second letter
print(str[3:5]) # from the third to fifth letter
print(str[-1]) # -1 means the last letter if you don't know the length
```

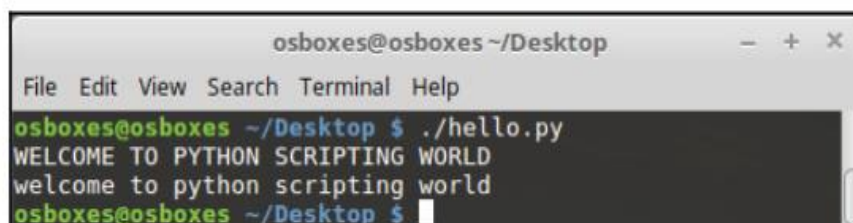
A terminal window titled 'likegeeks@likegeeks ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'likegeeks@likegeeks ~/Desktop \$'. The command './string-manipulation02.py' is entered and executed, resulting in the output: 'We', 'lcome to Python scripting world', 'co', 'd'. The prompt is now 'likegeeks@likegeeks ~/Desktop \$' with a cursor.

To replace a string, you can use the replace method like this:

```
#!/usr/bin/python3
str = "Welcome to Python scripting world"
str2 = str.replace("Python", "Shell")
print(str2)
```

A terminal window titled 'likegeeks@likegeeks ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'likegeeks@likegeeks ~/Desktop \$'. The command './string-manipulation03.py' is entered and executed, resulting in the output: 'Welcome to Shell scripting world'. The prompt is now 'likegeeks@likegeeks ~/Desktop \$' with a cursor.

To change the character case, you can use upper () and lower () functions:

A terminal window titled 'osboxes@osboxes ~/Desktop' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'osboxes@osboxes ~/Desktop \$'. The command './hello.py' is entered and executed, resulting in the output: 'WELCOME TO PYTHON SCRIPTING WORLD' and 'welcome to python scripting world'. The prompt is now 'osboxes@osboxes ~/Desktop \$' with a cursor.

Producing Scripts for Database, Web, and E-Mail: Writing database shell scripts-Using the Internet from your scripts-Emailing reports from scripts

Using Python as a Bash Scripting Alternative: Technical requirements-Python Language-Hello World the Python way-Pythonic arguments-Supplying arguments-Counting arguments-Significant whitespace-Reading user input-Using Python to write to files-String manipulation.

Unit Summary

This chapter walked through how to use some advanced features within your shell scripts. First, we discussed how to use the MySQL server to store persistent data for your applications. Just create a database and unique user account in MySQL for your application, and grant the user account privileges to only that database. You can then create tables to store the data that your application uses. The shell script uses the mysql command line tool to interface with the MySQL server, submit SELECT queries, and retrieve the results to display. Next we discussed how to use the lynxtext-based browser to extract data from websites on the Internet. The lynx tool can dump all the text from a web page, and you can use standard shell programming skills to store that data and search it for the content you're looking for. Finally, we walked through how to use the standard Mailx program to send reports using the Linux e-mail server installed on your Linux system. The Mailx program allows you to easily send output from commands to any e-mail address.

Let us sum up

The shell script uses the mysql command line tool to interface with the MySQL server, submit SELECT queries, and retrieve the results to display.

A major difference between Python and most other languages is that additional whitespace can mean something. The indent level of your code defines the block of code to which it belongs.

Check your progress

1. What is a primary advantage of using the dash shell over the bash shell?*

- A) More features
- B) Faster and smaller
- C) Easier syntax
- D) Better support for scripting

2. Which of the following is a characteristic of the dash shell?*

- A) It supports advanced scripting features.
- B) It is a non-POSIX-compliant shell.
- C) It is optimized for speed and low memory usage.
- D) It is the default shell on all Linux distributions.

3. What makes the zsh shell stand out compared to bash and dash?*

- A) It is simpler than both.
- B) It combines features from bash, ksh, and tcsh.
- C) It is the smallest shell available.
- D) It has no scripting capabilities.

4. Which of the following is a feature of zsh scripting?*

- A) Limited compatibility with other shells.
- B) Built-in support for floating-point arithmetic.
- C) Only supports basic scripting features.
- D) No support for arrays.

5. Which command is commonly used in shell scripts to create backups?*

- A) cp
- B) rm
- C) mv
- D) du

6. Which command would you use in a script to add a new user in Linux?*

- A) usermod B) useradd C) passwd D) chmod

7. What command can be used to check disk space usage in a script?*

- A) ls B) df C) du D) ps

8. Which command can be used to connect to a MySQL database from a shell script?*

- A) mysql B) dbconnect C) sqlconnect D) connectdb

9. Which command can be used to download a file from the internet in a shell script?*

- A) getfile B) wget C) fetch D) netget

10. Which command is commonly used to send emails from a shell script?*

- A) mail B) sendemail C) sendmail D) netmail

11. Which Python version introduced significant changes in syntax and features over the previous version?*

- A) Python 2.7 B) Python 1.6 C) Python 3.0 D) Python 3.9

12. What is the correct file extension for Python scripts?*

- A) .bash B) .py C) .sh D) .pyscript

13. What is the correct Python syntax to print "Hello World"?*

- A) echo "Hello World" B) print "Hello World"
C) printf("Hello World") D) print("Hello World")

14. Which module in Python is commonly used for parsing command-line arguments?*

- A) sys B) argparse C) getopt D) os

15. Which symbol is used to denote comments in Python?*

- A) # B) // C) /* */ D) <!-- -->

16. Which function is used to get number of command-line arguments in Python?*

- A) len(argv) B) argc C) len(sys.argv) D) argc(sys.argv)

17. What is the significance of whitespace in Python?*

- A) It is ignored. B) It separates commands.
C) It is used to denote blocks of code. D) It only matters in strings.

18. Which function is used to read input from the user in Python?*

- A) read() B) input() C) get() D) scanf()

19. Which mode is used to open a file for writing in Python?*

- A) "r" B) "rw" C) "w" D) "a"

20. Which method is used to convert a string to uppercase in Python?*

- A) upper() B) toUpperCase() C) to_upper() D) upcase()

Here are the answers:

1.B) 2.C) 3.B) 4.B) 5.A) 6.B) 7.B) 8.A) 9.B) 10.C) 11.B)
12.B) 13.D) 14.B) 15.A) 16.C) 17.C) 18.B) 19.C) 20.A)

Self Assessment Questions :

1. How dash shell is different from bash shell?
2. Summarize the basic features of Linux.
3. Compare dash shell and z-shell in advanced shell scripting.
4. Bash scripting using python in advanced shell scripting.
5. Write a shell script program to demonstrate to connect a PostgreSQL database and performing CRUD operations.
6. Give some example using python as a bash scripting.

Open source e-content links

1. Richard Blum, Christine Bresnahan, —Linux Command Line and Shell Scripting BIBLEll, Wiley Publishing, 3rd Edition, 2015.Chapters: 3, 11 to 14, 16 to 25.

2. Mokhtar Ebrahim, Andrew Mallett, —Mastering Linux Shell Scriptingll, Packt Publishing, 2nd Edition, 2018. Chapter: 14.

<https://www.javatpoint.com/linux-shell>

https://youtube.com/playlist?list=PLS1QuWo1RIaAsfcLW-Jk-Cx3JGRP8tjh&si=irjoFN25pPTosRX_

Glossary

python or python3: Invokes the Python interpreter.

Usage: python [options] [script] [args...]

`python -m`: Run a Python module as a script.

Usage: `python -m module [args...]`

Examples: `python -m http.server` (start a simple HTTP server)

`python -c`: Execute a Python command passed as a string.

Usage: `python -c 'command'`

Example: `python -c 'print("Hello, World!")'`

`python -i`: Start an interactive interpreter session after running a script.

Usage: `python -i script.py`

`str.replace()`: Replaces substrings in a string.

Usage: `new_string = original_string.replace('old', 'new')`

`str.split()`: Splits a string into a list.

Usage: `list_of_strings = string.split('delimiter')`

`str.join()`: Joins a list of strings into a single string.

Usage: `joined_string = delimiter.join(list_of_strings)`

`str.strip()`: Removes leading and trailing whitespace.

Usage: `clean_string = string.strip()`

e-books

1. Python for Unix and Linux System Administration by Noah Gift and Jeremy M. Jones
2. "Linux Shell Scripting with Bash" by Ken O. Burtch