Linux Fundamentals

Dr. Prashant Aparajeya

Learning Objectives:

- Understand what Linux is and why it is widely used.
- Become comfortable with the command-line interface (CLI) and basic navigation.
- Learn the Linux file system hierarchy and file permissions.
- Use package management (apt) to install, update, and manage software.
- Master advanced CLI techniques such as piping, redirection, and text processing.
- Monitor system resources and understand basic resource management commands.

Story 1: A Marine Biologist

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre, where she has been sampling gelatinous marine life in the Great Pacific Garbage Patch. She has 1520 samples that she's run through an assay machine to measure the relative abundance of 300 proteins. She needs to run these 1520 files through an imaginary program called `goostats.sh`. In addition to this huge task, she has to write up results by the end of the month, so her paper can appear in a special issue of *Aquatic Goo Letters*.

If Nelle chooses to run `goostats.sh` by hand using a GUI, she'll have to select and open a file 1520 times. If `goostats.sh` takes 30 seconds to run each file, the whole process will take more than 12 hours of Nelle's attention. With the shell, Nelle can instead assign her computer this mundane task while she focuses her attention on writing her paper.

The next few lessons will explore the ways Nelle can achieve this. More specifically, the lessons explain how she can use a command shell to run the `goostats.sh` program, using loops to automate the repetitive steps of entering file names, so that her computer can work while she writes her paper.

As a bonus, once she has put a processing pipeline together, she will be able to use it again whenever she collects more data.

Story 2: An Insurance Company

Fatma Yildiz, a dedicated data analyst at SecureLife Insurance, is responsible for processing incoming insurance claims. Each day, she receives hundreds of claims in CSV format, detailing information such as claim IDs, policy numbers, claimant names, claim amounts, and dates. Manually handling these files is time-consuming and prone to errors, which can lead to delays in claim settlements and customer dissatisfaction.

To streamline this process, Fatma decides to automate the extraction, validation, and loading of claims data into the company's database. By implementing a shell script, she can efficiently process each file, ensuring data accuracy and freeing up time to focus on more complex analytical tasks.

Module 1: Introduction to Linux & The Command Line

What is Linux?

- ★ Linux is an open-source, Unix-like operating system.
- ★ Key historical points:
  - Created by Linus Torvalds in 1991.
  - Combined with GNU utilities to form a complete OS.

GNU's Not Unix

GNU stands for "GNU's Not Unix." It is pronounced as one syllable with a hard g. Richard Stallman made the Initial Announcement of the GNU Project in September 1983. A longer version called the GNU Manifesto was published in March 1985.

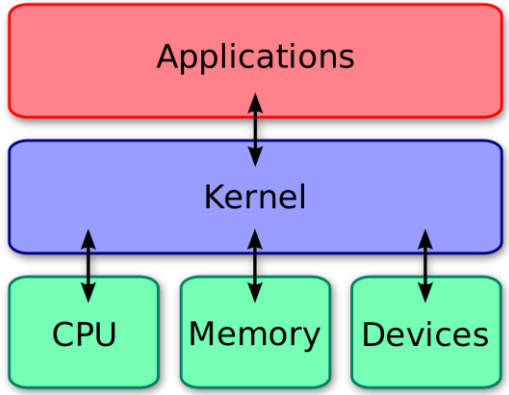
Linux began in 1991 as a personal project by Finnish student Linus Torvalds to create a new free operating system kernel. The resulting Linux kernel has been marked by constant growth throughout its history. Since the initial release of its source code in 1991, it has grown from a small number of C files under a license prohibiting commercial distribution to the 4.15 version in 2018 with more than 23.3 million lines of source code, not counting comments,[1] under the GNU General Public License v2 with a syscall exception meaning anything that uses the kernel via system calls are not subject to the GNU GPL.[2]:7[3][4]

Popular Linux distributions

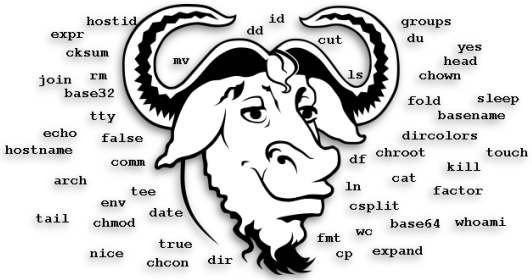
From sources across the web

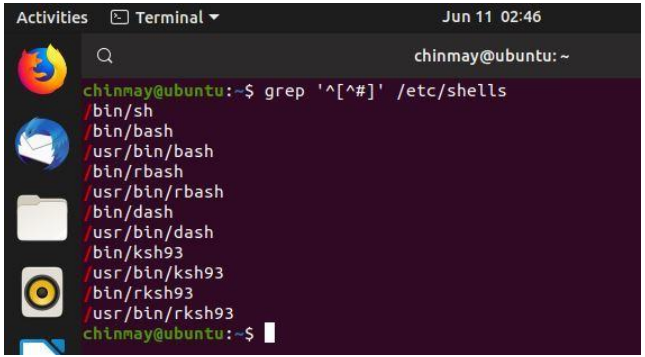
- Debian
- Linux Mint
- Arch Linux
- Ubuntu
- Manjaro
- CentOS
- Fedora
- openSUSE
- Red Hat Enterprise Linux

Linux Components



- The Linux Kernel: the core that manages hardware.
- GNU Utilities: the system programs and libraries.





- The Shell: the command-line interpreter (e.g., Bash).
- Graphical User Interface (GUI): optional layer on top of the OS.

Why Use the Command Line?

- ❏ Speed, automation, flexibility, and deeper control.
- ❏ Real-world examples: system administration, development, scripting.

Unix/Linux is user-friendly—it's just picky about who its friends are.

"In the beginning was the command line, and with it, the power to create whole new worlds from nothing but lines of code."

Opening a Terminal

How to launch the terminal in various environments:

- ❏ Ubuntu: using the “Activities” search or keyboard shortcut (Ctrl+Alt+T).
- ❏ Other Linux distributions and macOS (Terminal app) or Windows (WSL or Git Bash).
- ❏ When the shell is first opened, you are presented with a prompt, indicating that the shell is waiting for input.

Basic Commands – Navigation

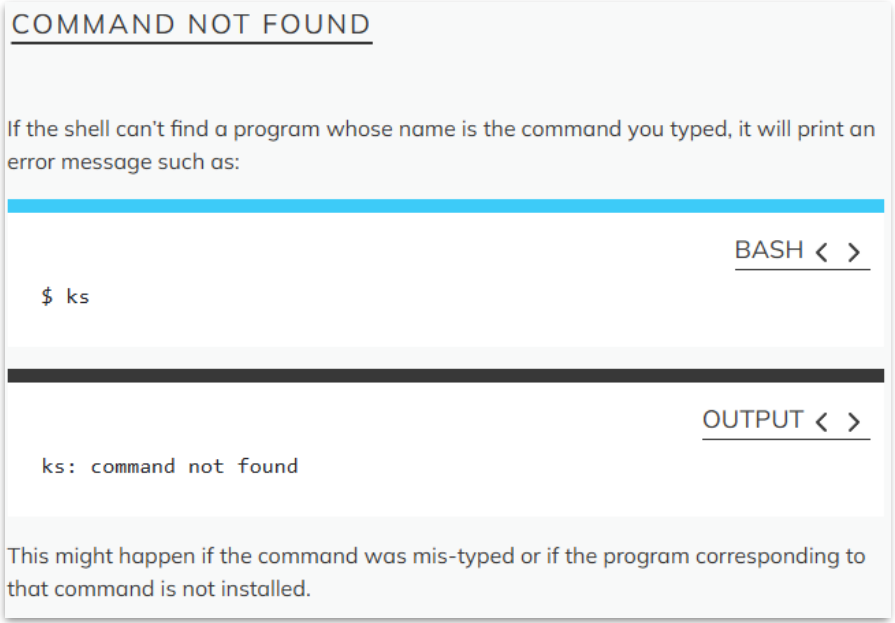
- `pwd`: “Print Working Directory.”
- `cd`: Change Directory.
  - Absolute path
  - Relative Path
- `ls`: List directory contents (with options like `-l`, `-F` and `-a`).

Command Manuals

- **`man` command (e.g., `man ls`) to look up command details.**
- Alternatively, `ls --help` can do the same job on Linux and Git Bash.

Troubleshooting

- ❏ Command not found
- ❏ Unsupported Command-Line Options (e.g. *ls -j*)
- ❏ No such file or directory



FUN TIME

1. Launch the Terminal
2. Run the Basic Commands and Note the output:
    a. pwd
    b. ls
    c. ls -l
    d. ls -a
3. Can you produce some unsupported or invalid options for ls?
4. Generate an error: "No such file or directory".
5. Explore for different ls options - Can you atleast identify **5** more options?

What did you learn in this Module?

- Launching a terminal,
- Navigating directories, and
- Running a few basic commands

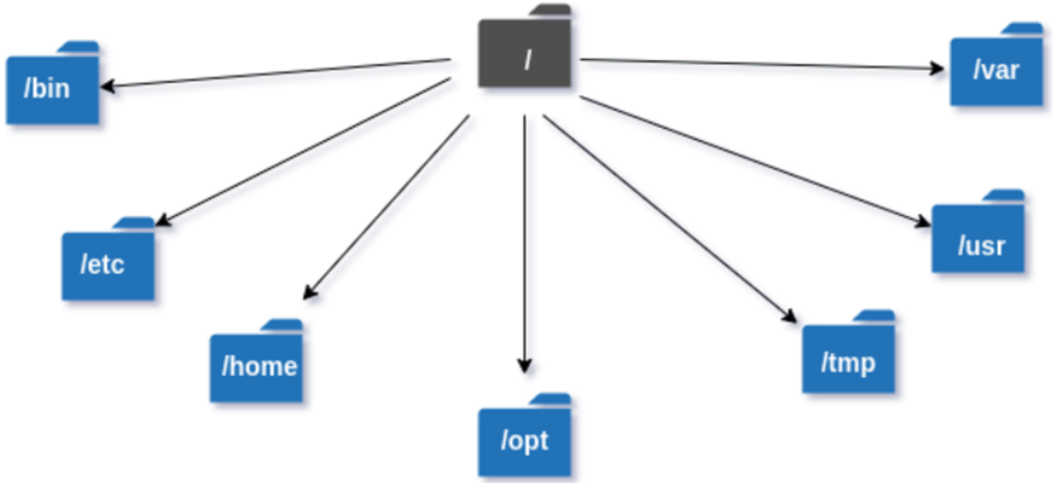
Key Points

- The file system is responsible for managing information on the disk.
- Information is stored in files, which are stored in directories (folders).
- Directories can also store other directories, which then form a directory tree.
- pwd prints the user's current working directory.
- `ls [path]` prints a listing of a specific file or directory; `ls` on its own lists the current working directory.
- `cd [path]` changes the current working directory.
- Most commands take options that begin with a single −.
- Directory names in a path are separated with / on Unix, but \ on Windows.
- / on its own is the root directory of the whole file system.
- An absolute path specifies a location from the root of the file system.
- A relative path specifies a location starting from the current location.
- `.` on its own means 'the current directory'; `..` means 'the directory above the current one'.

Module 2: Navigating the Linux File System and Basic File Operations

Understanding the Linux File System Hierarchy

We know that in a Windows-like operating system, files are stored in different folders on different data drives like C: D: E: whereas in the Linux/Unix operating system files are stored in a tree-like structure starting with the root directory as shown in the below diagram.



data storage in Linux/Unix operating systems

Root (/) Directory

- The root directory (/) is the highest level directory in a file system hierarchy, serving as the starting point from which all other files and directories branch out.
- Importance of the Root Directory:
  - **Central Hub**: The root directory functions as the central hub of the entire filesystem, providing access to every file and subdirectory.
  - **Hierarchical Organization**: It enables a systematic organization of files and directories, facilitating efficient data management and easy file location.
  - **Path Resolution:** All file and directory paths begin from the root, allowing for unique identification and minimizing naming conflicts.
  - **System Resources:** Critical system files and applications are often stored in directories directly under the root, such as /bin for binaries and /etc for configuration files.
  - **Starting Point for Navigation:** The root directory serves as the initial gateway for users and search engines to navigate the file system or website structure.

Overview of essential directories

- `/home`: User home directories.
- `/etc`: Configuration files.
- `/var`: Variable data like logs.
- `/usr`: User programs and shared resources.
- `/tmp`: Temporary files.

Absolute Paths

- An absolute path is a complete file or directory location in a file system, starting from the root directory and including all subdirectories leading to the specific file or folder.
- The key characteristics of an absolute path are:
  - It starts from the root directory (/)
  - It provides the full, unambiguous location of a file or directory
  - It remains the same regardless of the current working directory
  - It includes all directories in the hierarchy from root to the target
- Example: *"/home/user/documents/file.txt"*

Relative Paths

- A relative path is a way to specify the location of a file or directory in relation to the current working directory.
- Unlike an absolute path, a relative path does not start from the root directory and does not include the full path from the beginning of the file system hierarchy.
- Key characteristics of relative paths include:
  - They are based on the current working directory.
  - They do not start with a root directory symbol (/ in Unix-like systems or a drive letter in Windows).
  - They are typically shorter than absolute paths.
  - They can use special notations like "./" for the current directory and "../" for the parent directory.
- Example: “*../documents/file.txt*”
- Relative paths are commonly used in programming and web development for easier portability of code across different environments.
- They are particularly useful when working within a specific directory or subdirectory, as they allow for more concise file references.

Some `cd` options

`$ cd -` Toggles between last and present directory locations.

Now Try these options:

```
$ cd .
$ cd /
$ cd /home/{user_name}
$ cd ../..
$ cd ~
$ cd home
$ cd ~/data/..
$ cd
$ cd ..
```

Symbols

- `/` : Root directory.
- `~` : Current user’s home directory.
- `.` : Current directory.
- `..` : Parent directory.

Example

Absolute Paths

Absolute paths always start from the root directory (/):

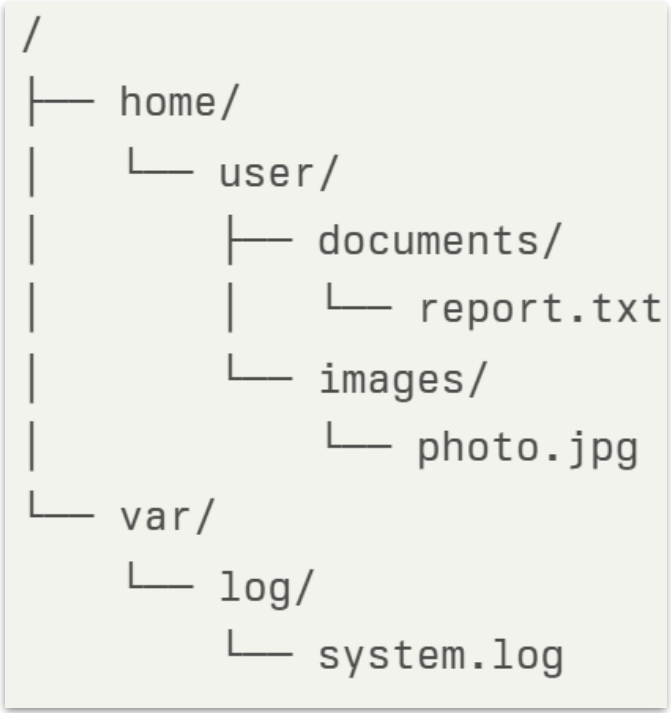
1. /home/user/documents/report.txt
2. /home/user/images/photo.jpg
3. /var/log/system.log

Relative Paths

Relative paths depend on the current working directory. Assuming we're in /home/user:

1. documents/report.txt
2. images/photo.jpg
3. ../var/log/system.log

The ".." notation means "go up one directory level".



File Types & Hidden Files

Regular Files

Regular files are the most common type of files in a file system. They contain data, such as text, images, or program code. Regular files:

- Store actual data content
- Can be read, written, or executed depending on permissions
- Are typically created and modified by users or applications

Symbolic Links

Symbolic links, also known as soft links, are special files that act as pointers to other files or directories. They:

- Contain a path to the target file or directory
- Can link to files or directories on different file systems
- Are denoted by an "l" in the first character of the file permissions when using `ls -l`

Hidden Files

Hidden files in Unix-like systems are easily recognizable:

- Their names start with a dot "." (e.g., .bashrc, .config)
- They are not displayed by default when using the `ls` command
- To view hidden files, use `ls -a` or `ls -la` in the terminal
- Hidden files are often used for storing user-specific configuration data

Basic Permissions Overview

File permissions in Unix-like systems control access to files and directories. The basic permission types are:

1. Read (r): Allows viewing file contents or listing directory contents
2. Write (w): Permits modifying or deleting files, or creating/deleting files within a directory
3. Execute (x): Enables running executable files or accessing directory contents

Permissions are assigned to three categories of users:

- Owner (u): The user who owns the file
- Group (g): Members of the file's group
- Others (o): All other users on the system

```
prashant@LAPTOP-T921THN8:~/repos/kaust/prashant/machine-learning$ ls -l
total 20
-rw-r--r-- 1 prashant prashant 11357 Dec 29 20:54 LICENSE
-rw-r--r-- 1 prashant prashant    18 Dec 29 20:54 README.md
drwxr-xr-x 4 prashant prashant  4096 Jan 15 20:04 notebook
-rw-r--r-- 1 prashant prashant     0 Dec 29 21:02 requirements.txt
```

Permissions can be viewed using the `ls -l` command, which displays them in symbolic notation (e.g., rwxr-xr-x) or numeric notation (e.g., 755).

Create, Copy, Rename Directories

mkdir Command

The mkdir (make directory) command is used to create new directories. Its basic syntax is:

```
mkdir [options] directory_name
```

Examples:

1. Create a directory in the current location:

   ```
   mkdir new_folder
   ```

2. Create a directory using an absolute path:

   ```
   mkdir /home/user/documents/project_folder
   ```

3. Create multiple directories at once:

   ```
   mkdir folder1 folder2 folder3
   ```

4. Create parent directories if they don't exist:

   ```
   mkdir -p /path/to/new/nested/directory
   ```

5. Set Directory Permissions

   ```
   mkdir -m 700 <directory>
   ```

6. Verbose Mode

   ```
   mkdir -v <directories>
   ```

Copy a Directory:

```
$ cp -r <source_dir> <dest_dir>
```

Rename a Directory:

```
$ mv <old_name> <new_name>
```

Removing Directories

rmdir Command

The rmdir (remove directory) command is used to delete new directories. Its basic syntax is:

```
rmdir [options] directory_name
```

Examples:

1. Remove an empty directory:
   ```
   rmdir empty_folder
   ```
2. Remove a directory using an absolute path:
   ```
   rmdir /home/user/documents/old_project
   ```
3. Remove parent directories if they become empty:
   ```
   rmdir -p /path/to/empty/nested/directory
   ```
4. Add safety with Prompts – Use the '`-i`' option to confirm each deletion:
   ```
   rm -ri <directory_to_delete>
   ```

Caution with Removing Directories

1. `rmdir` only removes empty directories. If a directory contains files or subdirectories, rmdir will fail.
2. For non-empty directories, the `rm` command with the `-r` (recursive) option can be used, but this should be done with extreme caution:
   ```
   rm -r directory_name
   ```
3. Always double-check the directory path before removing it, especially when using absolute paths or the recursive option.
4. Consider using the `-i` (interactive) option with rm to prompt for confirmation before each deletion.
5. Be particularly careful when removing directories with sudo privileges, as this can potentially damage your system if used incorrectly.

Remember, it's always a good practice to backup important data before performing operations that could result in data loss.

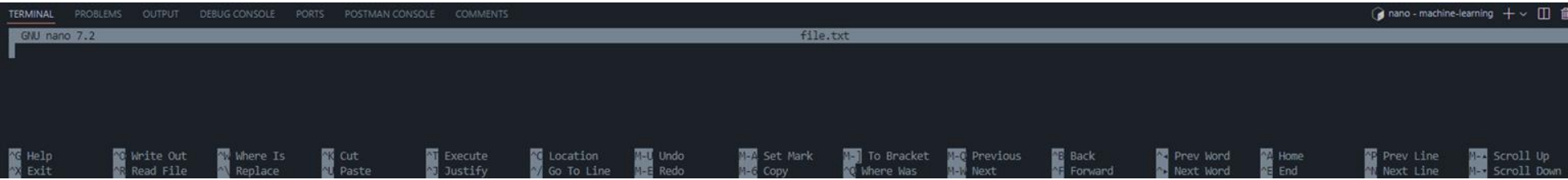
File Operations

- Create a text File:

  ```
  $ touch file.txt
  ```

  Alternatively, you can use `nano` editor if `nano` is installed on the system:

  ```
  $ nano file.txt
  ```



- Copy a File:

  ```
  $ cp <source_file> <destination_file>
  ```

  ```
  e.g.: $ cp file.txt copy_of_file.txt
  ```

- Move a File:

  ```
  $ mv file.txt /path/to/destination/
  ```

- Rename a File:

  ```
  $ mv file.txt new_file.txt
  ```

- Delete a File:

  ```
  $ rm file.txt
  ```

Wildcards

`*` is a wildcard, which represents zero or more other characters. Let's consider the `shell-lesson-data/exercise-data/alkanes` directory: `*.pdb` represents `ethane.pdb`, `propane.pdb`, and every file that ends with '.pdb'. On the other hand, `p*.pdb` only represents `pentane.pdb` and `propane.pdb`, because the 'p' at the front can only represent filenames that begin with the letter 'p'.

`?` is also a wildcard, but it represents exactly one character. So `?ethane.pdb` could represent `methane.pdb` whereas `*ethane.pdb` represents both `ethane.pdb` and `methane.pdb`.

Wildcards can be used in combination with each other. For example, `???ane.pdb` indicates three characters followed by `ane.pdb`, giving `cubane.pdb ethane.pdb octane.pdb`.

When the shell sees a wildcard, it expands the wildcard to create a list of matching filenames *before* running the preceding command. As an exception, if a wildcard expression does not match any file, Bash will pass the expression as an argument to the command as it is. For example, typing `ls *.pdf` in the `alkanes` directory (which contains only files with names ending with `.pdb`) results in an error message that there is no file called `*.pdf`. However, generally commands like `wc` and `ls` see the lists of file names matching these expressions, but not the wildcards themselves. It is the shell, not the other programs, that expands the wildcards.

List Filenames Matching a Pattern

When run in the `alkanes` directory, which `ls` command(s) will produce this output?

```
ethane.pdb methane.pdb
```

1. `ls *t*ane.pdb`
2. `ls *t?ne.*`
3. `ls *t??ne.pdb`
4. `ls ethane.*`

FUN TIME

1. List the "root" directory
2. Check your current location
3. Move to a directory using an "absolute" path
4. Do these:
    a. Change Directories:
        i. Go to "home" directory,
        ii. Go to "root" directory
        iii. Change to the /temp directory
    b. From the "root" directory:
        i. List the contents of "home" directory
        ii. Then list the contents on a directory in the "home" directory
        iii. Then change to that directory.
5. Display hidden files
6. Create, Explore and then Remove a "test" directory

What did you learn in this Module?

- How the Linux file system is organized,
- Be comfortable navigating through directories,
- Identifying hidden files, and
- Performing basic file operations

Key Points

- `cp [old] [new]` copies a file.
- `mkdir [path]` creates a new directory.
- `mv [old] [new]` moves (renames) a file or directory.
- `rm [path]` removes (deletes) a file.
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt`.
- `?` matches any single character in a filename, so `?.txt` matches `a.txt` but not any`.txt`.
- Use of the Control key may be described in many ways, including `Ctrl-X`, `Control-X`, and `^X`.
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Most files' names are `something.extension`. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Depending on the type of work you do, you may need a more powerful text editor than Nano.

Module 3: Package Management with `apt`

Introduction to Package Management

- Package management is a crucial process in software development that involves:
  - organizing,
  - controlling, and
  - distributing

  software components essential for servers and applications

Introduction to Package Management

- Package managers offer numerous benefits, including:
  - Control over software installation and removal
  - Visibility of installed packages
  - Security features like encryption and signing
  - Traceability of package versions and dependencies
  - Auditing capabilities for uploads and downloads

Introduction to Package Management

- A software package is a bundle of various software components and configuration files combined into a single unit. It typically contains:
  - Executable files
  - Configuration files
  - Libraries
  - Scripts
  - Documentation
- ***apt***'s role:
  - Retrieves package information from repositories.
  - Handles installation, upgrade, and removal of packages.

Essential `apt` Commands

- `sudo apt update` – Refresh package lists.
- `sudo apt upgrade` – Upgrade installed packages.
- `sudo apt install <package>` – Install a new package.
- `sudo apt remove <package>` – Remove an installed package.
- `sudo apt autoremove` – Remove orphaned dependencies.
- `apt-cache search <keyword>` – Search for packages.
    - `apt search <keyword>` – can also be used on the newer version of `apt`.

Using `sudo` with `apt`

Administrative privileges (sudo) are required when using apt for several important reasons:

1. System-wide changes: apt operations often modify system files and directories that are not accessible to regular users.
2. Package integrity: sudo ensures that only authorized users can install, update, or remove software packages, maintaining system security and stability.
3. Protection of critical components: Many packages managed by apt are essential for system functionality, and sudo prevents accidental or unauthorized modifications.

Using `sudo` with `apt`

When using sudo with apt, consider these safety tips:

1. Double-check commands before execution to avoid unintended changes.
2. Use specific sudo privileges for package management rather than full root access.
3. Regularly update the system to patch security vulnerabilities.
4. Be cautious when using commands like **`apt full-upgrade`**, which can remove installed packages.
5. Configure sudo to log all activities for auditing purposes.

Remember, with great power comes great responsibility.

Always exercise caution when using **`sudo`**, as it grants elevated privileges that can significantly impact your system if misused.

Best Practices & Troubleshooting

- Always run `apt update` before installing.
- Read the output carefully to catch any errors.
- When in doubt, check the manual pages (`man apt`).
- Discuss common issues such as repository errors or dependency conflicts.
- Regularly remove orphaned dependencies using `sudo apt autoremove`.

FUN TIME

1. Update Package Lists
2. Upgrade Installed Packages
3. Search for a Package - "apache2"
4. Package: "rolldice" or "tree" or "curl"
    a. Install this Package
    b. Verify the Installation
    c. Remove the Package
    d. Clear any dependencies that are no longer needed

FUN TIME

1. Update Package Lists

   `sudo apt update`

2. Upgrade Installed Packages

   `sudo apt upgrade`

3. Search for a Package - "curl"

   `apt search curl`

4. Package: "rolldice" or "tree" or "curl"
   a. `sudo apt install curl`
   b. `curl --version`
   c. `sudo apt remove curl`
   d. `sudo apt autoremove`

What did you learn in this Module?

- manage software on the Linux system,
- understand how dependencies are resolved, and
- appreciate best practices for maintaining a secure and updated system

Module 4: Advanced Command-Line Operations

Advanced Command-Line Operations

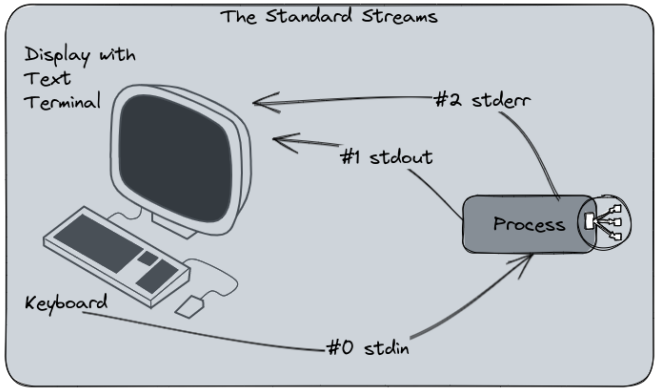
- ❏ Master input/output `redirection` techniques
- ❏ Utilize `pipes` to combine multiple commands effectively
- ❏ Employ `grep` for powerful text searching and pattern matching
- ❏ Leverage `sed` for efficient text manipulation and substitution
- ❏ Harness the power of `awk` for advanced text processing and data extraction

Understanding Standard Streams

Standard streams are fundamental input/output communication channels in Unix-like operating systems, including Linux. There are three standard streams:

- ❏ stdin (Standard Input, file descriptor 0): The input stream that provides data to commands.
- ❏ stdout (Standard Output, file descriptor 1): The output stream where commands send their normal output.
- ❏ stderr (Standard Error, file descriptor 2): The output stream for error messages and diagnostics.

```
 Keyboard
    |
    v
+--------+
| stdin  |
|   (0)  |
+--------+
    |
    v
+----------+
|  Command |
+----------+
    |
    |----------------+
    v                v
+--------+       +--------+
| stdout |       | stderr |
|   (1)  |       |   (2)  |
+--------+       +--------+
    |                |
    v                v
 Terminal         Terminal
(or File)        (or File)
```



In this diagram:

- stdin (0) receives input from the keyboard by default.
- The command processes the input and generates output.
- stdout (1) sends normal output to the terminal or a file.
- stderr (2) sends error messages to the terminal or a file.

By default, both stdout and stderr display their output on the terminal. However, these streams can be redirected to files or other commands using redirection operators (>, <, 2>) or pipes (|)

Example:

`$ ls > filelist.txt` # Redirect stdout to a file

`$ ls 2> error.log` # Redirect stderr to a file

Redirection Operators

Basic redirection:

- `>` : Overwrite a file with command output.

  ```
  $ echo "Hello, world!" > greeting.txt
  ```

- `>>`: Append command output to an existing file.

  ```
  $ echo "How are you?" >> greeting.txt
  ```

- `<` : Use a file as input to a command.

  ```
  $ sort < unsorted_list.txt
  ```

Special redirection:

- `2>&1`: Merge stderr with stdout.

  This redirects stderr (file descriptor 2) to the same destination as stdout (file descriptor 1):

  ```
  $ command > output.txt 2>&1
  ```

- Example: `find / -name "*.txt" > found_files.txt 2>&1`

Some Important Commands to Play with

`$ wc <file_name>` “word count” command to count the number of lines, words, and characters in files

`$ wc *.txt`

`$ wc -l *.txt` only number of lines are returned

`$ wc -l *.txt > lengths.txt` in the write mode it prints the output into lengths.txt (you can change this name)

`$ cat lengths.txt` concatenates together the file content and prints the contents of the files one after another.

`$ less lengths.txt` displays a screenful of the file, and then stops. Go forward - “space” backward - “b” quit - “q”

`$ sort -n lengths.txt` sorts the contents of the file (line wise operation)

`-n :` numerical sorting

`$ sort -n lengths.txt > sorted-lengths.txt`

`$ head -n 1 sorted-lengths.txt` tells it that we only want the first line of the file; -n 20 would get the first 20.

`$ head -n 3 animals.csv > animals-subset.csv`

`$ tail -n 2 animals.csv >> animals-subset.csv`

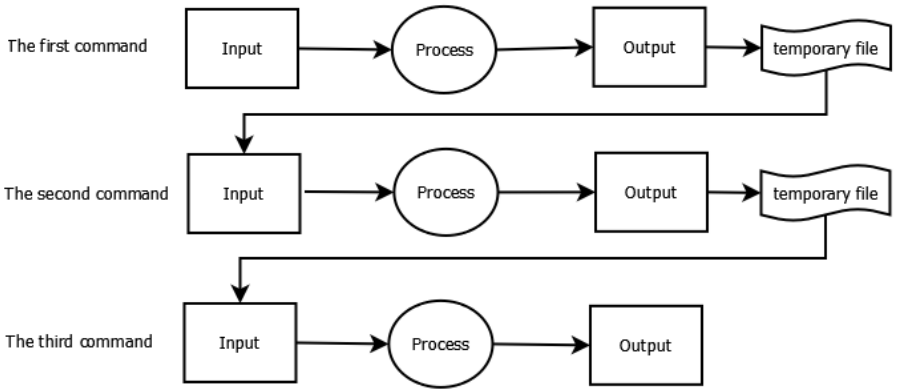
`$ cut -d , -f 2 animals.csv` used to remove or ‘cut-out’ certain sections of each line in the file
`-d` delimiter for field delimiter
`-f` select only these fields; also print any line that contains no delimiter character, unless the -s option is specified

Introduction to Piping

- The pipe operator (|) in Linux is a powerful tool that allows you to chain multiple commands together by feeding the output of one command as input to the next. This enables the creation of complex command sequences and efficient data processing on the command line.

```
$ ls | grep
$ cat logfile.txt |
$ sort -n lengths.t
$ wc -l *.
$ wc -l *.txt |
```

- Exa




```
$ cat animals.csv | head -n 5 | tail -n 3 | sort -r > final.txt
```

Text Processing with `grep`

- `grep` is a powerful command-line utility for searching and filtering text based on patterns. It's an essential tool for text processing in Unix-like operating systems.
- `grep` searches input files for lines containing a match to a given pattern. Its basic syntax is:

  ```
  $ grep [options] pattern [file...]
  ```
- Key features of grep include:
  - Pattern matching using regular expressions
  - Filtering lines from files or command output
  - Displaying matching lines or file names

Common `grep` Options

1. -i (ignore case): Performs case-insensitive matching

   ```
   $ grep -i "gnu" GPL-3
   ```

   This command finds all occurrences of "gnu", "GNU", "Gnu", etc. in the GPL-3 file.
2. -r (recursive search): Searches files in the current directory and all subdirectories

   ```
   $ grep -r "pattern" /path/to/directory
   ```
3. -v (invert match): Displays lines that do not match the pattern

   ```
   $ grep -v "string" file.txt
   ```

   This returns all lines that do not contain "string".
4. -n (line number): Displays the line number along with the matching lines

   ```
   $ grep -n "pattern" file.txt
   ```
5. -l (files with matches): Shows only the names of files containing matches

   ```
   $ grep -l "pattern" *
   ```
6. -c (count): Displays the count of matching lines instead of the lines themselves

   ```
   $ grep -c "pattern" file.txt
   ```

grep is highly versatile and can be combined with other commands using pipes (|) for complex text processing tasks. For example:

```
$ ls | grep ".txt" | wc -l
```

Using `sed` for Stream Editing

`sed` (stream editor) is a powerful command-line utility for text processing in Unix-like operating systems. It's particularly useful for substitution and deletion operations on text streams or files.

Substitution with `sed`

- The basic syntax for substitution in sed is:

```
sed 's/pattern/replacement/flags'
```

- To substitute "error" with "warning" globally in a file:

```
sed 's/error/warning/g' input_file.txt
```

This command reads input_file.txt, replaces all occurrences of "error" with "warning", and outputs the result to the terminal. The 'g' flag ensures all matches on each line are replaced.

Deletion with `sed`

- To delete lines matching a pattern:

```
sed '/pattern/d' input_file.txt
```

- For example, to delete all lines containing "error":

```
sed '/error/d' input_file.txt
```

Key Features of `sed`

1. In-place editing: Use the -i option to edit files directly.
2. Multiple commands: Combine commands using semicolons or -e option.
3. Regular expressions: Utilize powerful pattern matching.
4. Address ranges: Apply commands to specific line ranges.

sed is particularly efficient for processing large files or streams of text, as it operates line by line without loading the entire file into memory

Introduction to `awk`

- `awk` is a powerful text processing tool used for pattern scanning and processing in Unix-like operating systems. It reads input line by line, searching for patterns, and performs specified actions when matches are found.

- Basic syntax of awk:

```
awk 'pattern { action }' input_file
```

Key features of awk include:

1. Pattern matching using regular expressions
2. Processing text line by line
3. Performing actions on matched patterns
4. Built-in variables for field and record manipulation

Common use cases

1. Printing specific columns:
   ```
   awk '{print $3 "\t" $4}' file.txt
   ```
   This prints the 3rd and 4th columns of file.txt, separated by a tab[1].

1. Filtering lines based on conditions:
   ```
   awk '$3 > 15 {print $1}' file.txt
   ```
   This prints the 1st column for lines where the 3rd column is greater than 15[1].

1. Counting occurrences:
   ```
   awk '/pattern/{++cnt} END {print "Count = ", cnt}' file.txt
   ```
   This counts and prints the number of lines matching a pattern.

1. Formatting output:
   ```
   awk '{printf "%-10s %s\n", $2, $3}' file.txt
   ```
   This prints the 2nd and 3rd columns with formatted spacing.

`awk` is particularly useful for processing structured text files, such as CSV or tab-separated files, and for extracting specific data from log files or other text-based outputs.

`awk` comparison with `grep` and `sed`

`awk` is a powerful text processing tool that combines the filtering capabilities of `grep` with the text manipulation features of `sed`, while also offering additional formatting and computational abilities. Here's a comparison of how `awk` can combine filtering and formatting in ways that `grep` and `sed` cannot:

1. Filtering:
   - grep: `grep "pattern" file.txt`
   - awk: `awk '/pattern/' file.txt`
2. Filtering with column-based conditions:
   - sed: Requires complex expressions
   - awk: `awk '$3 > 100 {print $1, $2}' file.txt`
3. Combining filtering and formatting:
   - grep + sed: Requires piping multiple commands
   - awk: `awk '$3 > 100 {printf "%-10s %s\n", $1, $2}' file.txt`
4. Performing calculations while filtering:
   - grep + sed: Not directly possible
   - awk: `awk '$3 > 100 {sum += $4} END {print "Total:", sum}' file.txt`

`awk`'s ability to process structured data, perform calculations, and format output makes it a versatile tool for complex text processing tasks that would require multiple steps with `grep` and `sed`

Chaining Commands Together

- Let's walk through a workflow that processes web server log files to extract useful information:

1. `grep "ERROR" /var/log/apache2/error.log | awk '{print $4, $5, $NF}' | sort | uniq -c | sort -nr | head -n 5 > top_errors.txt`

2. `cat access.log | grep "404" | awk '{print $1, $4}' | sort | uniq -c > not_found_summary.txt`

This command chain performs the following steps:

1. `grep "ERROR" /var/log/apache2/error.log`: Filters the Apache error log for lines containing "ERROR".
2. `awk '{print $4, $5, $NF}'`: Extracts the date, time, and error message from each line.
3. `sort`: Sorts the extracted information alphabetically.
4. `uniq -c`: Counts the occurrences of each unique error.
5. `sort -nr`: Sorts the errors numerically in reverse order (most frequent first).
6. `head -n 5`: Selects the top 5 most frequent errors.
7. `> top_errors.txt`: Redirects the final output to a file named top_errors.txt.

This workflow demonstrates how multiple commands can be chained together using pipes (|) to process data step by step, with the final output redirected to a file. It combines text filtering (grep), data extraction (awk), sorting, counting, and output limitation to analyze log files efficiently.

FUN TIME

1. Prepare a Sample File:
    a. Create a file named sample.txt with several lines of text (or use an existing log file).
2. Redirection Practice:
    a. Redirect the contents of sample.txt to another file
    b. Append additional text to this new file
3. Using grep:
    a. Filter lines containing "error" (case insensitive)
4. Using sed:
    a. Replace the word "error" with "ERROR" in the output (do not modify the file)
    b. Optionally, write the changes to a new file
5. Using awk:
    a. Assuming your sample file has multiple fields (you might add sample fields), extract the first word of each line
6. Chain Operations:
    a. Combine the commands to, for example, count the number of lines containing "ERROR"

FUN TIME

1. Prepare a Sample File:

   ```
   echo -e "Line one: success\nLine two: ERROR occurred\nLine three: warning\nLine four: error detected" > sample.txt
   ```

2. Redirection Practice:
   a. `cat sample.txt > copy.txt`
   b. `echo "Line five: additional error" >> copy.txt`
3. Using grep:
   a. `grep -i "error" copy.txt`
4. Using sed:
   a. `sed 's/error/ERROR/g' copy.txt`
   b. `sed 's/error/ERROR/g' copy.txt > modified.txt`
5. Using awk:
   a. `awk '{print $1}' copy.txt`
6. Chain Operations:
   a. `grep -i "error" copy.txt | sed 's/error/ERROR/g' | wc -l`

What did you learn in this Module?

- redirect output,
- pipe commands together, and
- utilize text processing tools to filter and transform data

Module 5: Loops

Role of Loops in Shell Scripting

Loops automate repetitive tasks by executing commands:

- Process multiple files/directories
- Repeat actions until conditions are met
- Handle large datasets efficiently
- Create scheduled/background tasks

for Loop

Iterates over a list of items:

```
for item in list; do
  [commands]
done
```

Iterates over a list of commands.

Example 1: Iterate over files

```
for file in *.txt; do
  echo "Processing $file"
  wc -l "$file"
done
```

Example 2: Iterate over command output

```
for user in $(cut -d: -f1 /etc/passwd); do
  echo "User: $user"
done
```

Bulk Renaming Files:

```
counter=1
for file in *.jpg; do
  mv "$file" "vacation_$(printf "%03d" $counter).jpg"
  ((counter++))
done
```

`while` Loop

Runs while a condition is true:

```
while [condition]; do
  [commands]
done
```

Iterates over a list of commands.

Example : Countdown timer

```
count=5
while [ $count -gt 0 ]; do
  echo "$count..."
  sleep 1
  ((count--))
done
echo "Liftoff!"
```

Monitoring System:

Alert when disk usage exceeds 90%

```
while true; do
  usage=$(df / | awk 'END{print $5}' | tr -d '%')
  [ $usage -gt 90 ] && echo "ALERT: Disk full!" |
mail -s "Disk Warning" admin@example.com
  sleep 3600 # Check hourly
done
```

`until` Loop

Runs until a condition becomes true:

```
until [condition]; do
  [commands]
done
```

Iterates over a list of commands.

Example : Countdown timer

```
count=5
while [ $count -gt 0 ]; do
  echo "$count..."
  sleep 1
  ((count--))
done
echo "Liftoff!"
```

Key Differences Between Loop Types

Loop Type	Best For	Termination Condition
`for`	Known iterations	End of list
`while`	Unknown iterations	When condition becomes false
`until`	Wait for condition to be met	When condition becomes true

Advanced Loop Techniques

Nested Loops:

```
for i in {1..3}; do
  for j in {a..c}; do
    echo "Pair: $i$j"
  done
done
```

Loop Control:

- `break`: Exit loop immediately
- `continue`: Skip to next iteration

```
for num in {1..10}; do
  [ $num -eq 5 ] && break
  [ $num -eq 3 ] && continue
  echo $num
done
```

Infinite Loops:

```
while :; do
  echo "Running forever..."
  sleep 1
done
```

FUN TIME

1. Write a simple bash script that processes files in a directory and counts the number of lines in each .txt files

Best Practices

1. Quote variables: `"$file"` handles spaces in filenames
2. Test with `echo` first: Dry-run before destructive commands
3. Use `$(command)` syntax: For command substitution
4. Limit recursion: Avoid infinite loops in `while`/`until`
5. Add sleep intervals: For resource-intensive loops

What did you learn in this Module?

- how loops function in Bash,
- be able to write basic shell scripts incorporating loops, and
- appreciate the power of automation in the Linux environment

Key Points

- A `for` loop repeats commands once for every thing in a list.
- Every `for` loop needs a variable to refer to the thing it is currently operating on.
- Use `$name` to expand a variable (i.e., get its value). `${name}` can also be used.
- Do not use spaces, quotes, or wildcard characters such as '*' or '?' in filenames, as it complicates variable expansion.
- Give files consistent names that are easy to match with wildcard patterns to make it easy to select them for looping.
- Use the up-arrow key to scroll up through previous commands to edit and repeat them.
- Use Ctrl+R to search through the previously entered commands.
- Use `history` to display recent commands, and `![number]` to repeat a command by number.

Module 6: System Monitoring & Resource Management

Why System Monitoring Matters

System monitoring is crucial for:

- Early Problem Detection: Identify issues before they cause downtime
- Performance Optimization: Pinpoint bottlenecks (CPU, RAM, disk I/O)
- Capacity Planning: Understand resource needs for scaling
- Security Monitoring: Detect suspicious activity/processes
- Cost Management: Right-size cloud/hardware resources
- Service Availability: Ensure critical services remain responsive

Essential Monitoring Tools

`top` - Real-Time Process Overview

`htop` - Enhanced Interactive Viewer

`free` - Memory Utilization (check with options `-m` and `-g`)

`df`- Disk space monitoring

`vmstat` - System-Wide Statistics (e.g. `vmstat 2 5`)

Displays system statistics on processes, memory, paging, block I/O, traps, and CPU activity. Running it with parameters (e.g., 2 5) will show updates every 2 seconds for 5 iterations.

Critical Metrics to Monitor

Metric	Healthy Range	Warning Signs
CPU Load Average	< Number of cores	Sustained > 2x cores
Memory Usage	< 80% of total	High swap usage (si/so > 0)
Disk I/O Wait	< 5%	Sustained > 20%
Context Switches	Depends on workload	Sudden spikes
Network Utilization	< 70% of bandwidth	Packet drops/errors

Logging & Analysis Techniques

Basic Logging with `vmstat`

```
$ vmstat 60 > system_stats.log &  # Log every 60 seconds
```

Basic Logging with `top`

```
$ top -b -n 1 > top_snapshot.txt
```

Scheduled Monitoring with Cron

```
# Add to crontab (crontab -e)

$ */5 * * * * /usr/bin/top -b -n1 | head -20 > /var/log/top_snapshot.log
```

Quick Health Check

```
$ echo "=== $(date) ===";

$ echo "Load: $(uptime)";

$ free -h | awk '/Mem/{print "Memory:", $3"/"$2}';

$ df -h / | tail -1 | awk '{print "Disk:", $3"/"$2}'
```

What did you learn in this Module?

- monitor real-time system performance,
- understand the basic metrics displayed, and
- know how to capture and log this information for further analysis

Story 1: A Marine Biologist

Nelle Nemo, a marine biologist, has just returned from a six-month survey of the North Pacific Gyre, where she has been sampling gelatinous marine life in the Great Pacific Garbage Patch. She has 1520 samples that she's run through an assay machine to measure the relative abundance of 300 proteins. She needs to run these 1520 files through an imaginary program called `goostats.sh`. In addition to this huge task, she has to write up results by the end of the month, so her paper can appear in a special issue of *Aquatic Goo Letters*.

If Nelle chooses to run `goostats.sh` by hand using a GUI, she'll have to select and open a file 1520 times. If `goostats.sh` takes 30 seconds to run each file, the whole process will take more than 12 hours of Nelle's attention. With the shell, Nelle can instead assign her computer this mundane task while she focuses her attention on writing her paper.

The next few lessons will explore the ways Nelle can achieve this. More specifically, the lessons explain how she can use a command shell to run the `goostats.sh` program, using loops to automate the repetitive steps of entering file names, so that her computer can work while she writes her paper.

As a bonus, once she has put a processing pipeline together, she will be able to use it again whenever she collects more data.

Story 1: Solution

bash

Copy Edit

```bash
#!/bin/bash

# Directory containing the sample files
DATA_DIR="data"

# Output directory for processed results
OUTPUT_DIR="results"

# Ensure the output directory exists
mkdir -p "$OUTPUT_DIR"

# Loop through all files in the data directory
for file in "$DATA_DIR"/*.txt; do
    # Extract filename without path
    filename=$(basename "$file")

    # Run the analysis and save the output
    ./goostats.sh "$file" > "$OUTPUT_DIR/${filename%.txt}_stats.txt"

    echo "Processed $file"
done

echo "All files have been processed!"
```

Explanation

1. **Defines Directories:**
   - `DATA_DIR` contains the 1520 input files.
   - `OUTPUT_DIR` stores the processed results.
   - The script ensures that the output directory exists (`mkdir -p "$OUTPUT_DIR"`).
2. **Loops Over All Files:**
   - It finds all `.txt` files in `DATA_DIR`.
   - Extracts the filename using `basename`.
3. **Runs `goostats.sh` on Each File:**
   - The output is saved with `_stats.txt` suffix in `OUTPUT_DIR`.
4. **Displays Progress Messages:**
   - `echo "Processed $file"` informs the user of progress.
   - Once all files are processed, a completion message is printed.

How to Use

1. Save this script as `process_samples.sh`.
2. Give it execute permission:

   ```
   chmod +x process_samples.sh
   ```

3. Run it:

   ```
   ./process_samples.sh
   ```

This script will allow Nelle to focus on her paper while the computer processes her data!

Story 2: An Insurance Company

Fatma Yildiz, a dedicated data analyst at SecureLife Insurance, is responsible for processing incoming insurance claims. Each day, she receives hundreds of claims in CSV format, detailing information such as claim IDs, policy numbers, claimant names, claim amounts, and dates. Manually handling these files is time-consuming and prone to errors, which can lead to delays in claim settlements and customer dissatisfaction.

To streamline this process, Fatma decides to automate the extraction, validation, and loading of claims data into the company's database. By implementing a shell script, she can efficiently process each file, ensuring data accuracy and freeing up time to focus on more complex analytical tasks.

Story 2: Solution

```bash
#!/bin/bash

# Directory containing incoming claims files
CLAIMS_DIR="claims_data"

# Directory to archive processed files
ARCHIVE_DIR="processed_claims"

# Log file for processing details
LOG_FILE="claims_processing.log"

# Ensure the archive directory exists
mkdir -p "$ARCHIVE_DIR"

# Loop through all CSV files in the claims directory
for file in "$CLAIMS_DIR"/*.csv; do
    # Extract filename without path
    filename=$(basename "$file")

    # Log the start of processing
    echo "$(date): Starting processing of $filename" >> "$LOG_FILE"

    # Extract: Read data from the CSV file
    while IFS=, read -r claim_id policy_number claimant_name claim_amount claim_date; do
        # Transform: Perform data validation and formatting
        # Example: Check if claim_amount is a valid number
        if ! [[ "$claim_amount" =~ ^[0-9]+(\.[0-9]{1,2})?$ ]]; then
            echo "$(date): Invalid claim amount in $filename: $claim_amount" >> "$LOG_FILE"
            continue
        fi

        # Load: Insert the data into the database (pseudo-code)
        # Replace the following line with actual database insertion command
        echo "INSERT INTO claims (claim_id, policy_number, claimant_name, claim_amount, claim_date) VALUES
        ('$claim_id', '$policy_number', '$claimant_name', '$claim_amount', '$claim_date');" >> "$LOG_FILE"

    done < "$file"

    # Move the processed file to the archive directory
    mv "$file" "$ARCHIVE_DIR/"

    # Log the completion of processing
    echo "$(date): Completed processing of $filename" >> "$LOG_FILE"
done

echo "All claim files have been processed!"
```

Explanation

- **Directories:**
  - `CLAIMS_DIR`: Contains the incoming claims CSV files.
  - `ARCHIVE_DIR`: Stores the processed files for record-keeping.
- **Log File:**
  - `LOG_FILE`: Records processing activities and any issues encountered.
- **Processing Loop:**
  - The script iterates through each CSV file in the `CLAIMS_DIR`.
  - For each file, it reads the data line by line.
  - **Data Validation:** Checks if the `claim_amount` is a valid number.
  - **Data Loading:** Inserts the validated data into the database. (Note: The actual database insertion command should replace the placeholder echo statement.)
  - After processing, the file is moved to the `ARCHIVE_DIR`.

How to Use

1. **Save the Script:** Save the script as `process_claims.sh`.
2. **Make it Executable:** Run `chmod +x process_claims.sh` to make the script executable.
3. **Execute the Script:** Run `./process_claims.sh` to start processing the claims files.

Practice Coding – A LOT

Any Questions