SHELL SCRIPTING

What are shells?

A shell is a command-line interface (CLI) program that provides a way for users to interact with an operating system's kernel and perform various tasks using text-based commands. The shell takes user inputs, interprets them, and communicates with the operating system to execute the requested actions.

In essence, a shell acts as an intermediary between the user and the underlying operating system, allowing users to interact with the system's resources, files, processes, and services through text-based commands. Shells are an essential part of Unix-like operating systems, including Linux.

Characteristics and functions of a shell:

Command Interpretation: The primary role of a shell is to interpret the commands entered by the user. It understands the commands' syntax and semantics and translates them into actions that the operating system can perform.

Execution of Programs: Shells can run various types of programs, from system utilities to user-written scripts. When you enter a command, the shell starts the appropriate program or executes the built-in command.

Input and Output Redirection: Shells allow users to redirect input and output streams. This means you can send the output of a command to a file or read input from a file instead of the keyboard.

Scripting: Shells support scripting, which involves writing sequences of commands in a script file. These scripts can be executed as a single command, making it easy to automate repetitive tasks.

Environment Variables: Shells maintain environment variables, which are values that control various aspects of the shell and the system. These variables can affect behavior, configuration, and settings.

Control Structures: Shells offer control structures like loops and conditional statements, which allow you to create more complex and dynamic command sequences.

Customization: Users can customize their shell environment by setting variables, defining aliases (shortcuts for commands), and creating shell configuration files.

Common shells in the Unix-like environment include:

Bash (Bourne Again Shell): The default shell for many Linux distributions, known for its wide adoption and rich features.

Zsh (Z Shell): An extended version of Bash with additional features, improved auto-completion, and customization options.

Fish (Friendly Interactive Shell): A user-friendly shell with advanced auto-suggestions and syntax highlighting.

tcsh: An enhanced version of the C shell (csh), offering more advanced interactive features.

Shells provide users with a powerful and flexible way to interact with the system, manage files, run programs, and automate tasks.

Shell Variables Scriptings

Shell variables are placeholders used in shell scripting to store and manipulate data, making scripts more flexible and dynamic. They allow you to store values, retrieve them, and perform operations based on these values. Shell scripting involves using variables to enhance the functionality of your scripts.

Here are some key points about shell variables and shell scripting:

Variable Assignment: You can assign values to variables using the syntax variable_name=value. No spaces are allowed around the equals sign.

Accessing Variable Values: To access the value stored in a variable, use the syntax $variable_name or ${variable_name}.

Types of Variables:

User-Defined Variables: Created and used by the script writer.

System Variables: Predefined variables that hold information about the system environment.

Special Variables: Shell provides a set of predefined variables for specific purposes, like $0 (script name), $1, $2, ... (command-line arguments), and more.

Variable Expansion: Variables are expanded within double-quoted strings, allow to include their values in output.

Arithmetic Operations: Shell variables can hold numeric values, and you can perform arithmetic operations using various syntaxes, like expr, $(( )), or let.

Concatenation: You can concatenate strings using variables, e.g., result="$variable1$variable2".

Read Input: Use the read command to accept user input and store it in variables.

Scope: Variables have local scope by default. To make them global (visible outside a function), you need to use the export command.

Substitution: Variables can be used for command substitution. For example, output=$(ls) stores the output of the ls command in the output variable.

Default Values:Set default values for variables using ${variable_name:-default_value}.

Arrays: Some shells support arrays, allow to store multiple values under a single variable.

Simple example of using variables in a Bash script:

```bash
#!/bin/bash

name="John"
age=30

echo "My name is $name and I am $age years old."
```

This script uses variables to store a name and an age and then displays them using the echo command.

Shell variables enhance the capabilities of shell scripting by allowing you to create dynamic, reusable, and interactive scripts. They provide the means to store, manipulate, and use data within your scripts effectively.

Combining Shell Commands In Linux

In Linux, you can combine multiple shell commands to perform more complex tasks using various techniques. This allows you to create powerful one-liners or scripts that accomplish multiple actions in a single command line. Here are some ways to combine shell commands:

Pipeline (|): The pipeline operator allows you to send the output of one command as the input to another. This is useful for chaining commands together.

```
command1 | command2
```

For example, to list all files in a directory and then count the number of files, you can use:

```
ls -l | wc -l
```

Command Substitution ($() or ``)`: Command substitution lets you use the output of one command as an argument for another.

```bash
variable=$(command)
```

For instance, to store the output of the date command in a variable, you can use:

```bash
current_date=$(date)
```

Sequential Execution (;): You can use a semicolon to execute commands sequentially, one after another.

```bash
command1 ; command2
```

For example, to create a new directory and then change into that directory, you can use

```bash
mkdir new_directory ; cd new_directory
```

Logical Operators (&& and ||): These operators allow you to run the second command only if the first one succeeds (&&) or fails (||).

```bash
command1 && command2
command1 || command2
```

To remove a file if it exists and then create a new file, you can use:

```bash
rm file.txt && touch file.txt
```

Grouping Commands (()): Parentheses are used to group commands, enabling you to apply operations to the group as a whole.

```bash
(command1 ; command2) ; command3
```

For example, to change to a directory, run a command, and then return to the previous directory, you can use:

```bash
(cd directory ; command ; cd -)
```

These techniques allow to create complex command sequences that perform multiple tasks in a single line. Combining commands in Linux is an efficient way to streamline your workflow and achieve more with fewer lines of code.

Editing

Editing in shell scripting refers to modifying text files or scripts using command-line tools available in the shell environment. Shell scripts are collections of commands written in a scripting language (like Bash) that automate tasks.

How editing works in shell scripting:

- **Text Editors**: Shell scripts often involve creating or modifying text files. You can use text editors like "nano," "vim," or "emacs" directly from the command line to open and edit files.
- **Creating New Files:** To create a new file, you use a text editor with the file name as an argument. For example, nano newfile.txt will open a new file named "newfile.txt" in the "nano" editor.
- **Modifying Existing Files:** To modify an existing file, you use a text editor with the file name as an argument. For instance, nano

existingfile.txt will open the "existingfile.txt" file in the "nano" editor for editing.

- **Editing Content**: Once the file is open in the text editor, you can navigate through the file using keyboard shortcuts, make changes to the content, and save the changes.
- **Saving Changes**: After making changes, save the modified file. In "nano," you might press "Ctrl + O" to write the changes to the file and "Ctrl + X" to exit the editor.
- **Shell Commands for Editing**: Besides using text editors, shell scripting also allows you to manipulate text files using various commands. For example:
- **echo:** Appends text to a file or displays text on the screen.
- **cat:** Displays the content of a file.
- **sed and awk**: Text processing tools for finding, replacing, and manipulating text within files.
- **grep**: Searches for specific patterns in files.
- **Variable Substitution**: Within shell scripts, you can use variables to store and manipulate text. Variables are placeholders that can be replaced with their values.
- **Here Documents:** Shell scripts often use "here documents" to include multi-line content within a script without needing to create separate files.
- **Permissions:** Remember to consider file permissions when editing files. Make sure you have the necessary permissions to read and write to the files you're editing.

Shell scripting and editing are crucial skills for automating tasks and managing configurations in a command-line environment

Overview of vi

An overview of the vi text editor. vi is a powerful and widely used text editor found on Unix-based systems. It is known for its efficiency and versatility, although its interface might be initially challenging for new users. Here's a brief overview of vi:

1. Modes:

vi operates in different modes: Normal mode, Insert mode, and Command-line mode.

Normal mode: This is the default mode where you can navigate and manipulate text.

Insert mode: In this mode, you can directly type and edit text.

Command-line mode: Used for entering commands like saving, quitting, and searching.

2. Basic Navigation in Normal Mode:

‘h’: Move cursor left.

‘j’: Move cursor down.

‘k’: Move cursor up.

‘l’: Move cursor right.

‘w’: Move to the beginning of the next word.

‘b’: Move to the beginning of the previous word.

‘0’: Move to the beginning of the current line.

‘$’: Move to the end of the current line.

‘gg’: Move to the beginning of the file.

‘G’: Move to the end of the file.

‘:{line_number}’: Move to a specific line number.

3. Editing in Normal Mode:

‘x’: Delete the character under the cursor.

‘dd’: Delete the current line.

‘yy’: Copy the current line.

‘p’: Paste the copied or deleted content.

‘u’: Undo the last action.

‘Ctrl + r’: Redo.

4. Inserting and Editing in Insert Mode:

Press ‘i’ in Normal mode to enter Insert mode at the cursor.

Press ‘a’ in Normal mode to enter Insert mode after the cursor.

Press ‘o’ in Normal mode to open a new line below and enter Insert mode.

Press ‘O’ in Normal mode to open a new line above and enter Insert mode.

Press ‘Esc’ to return to Normal mode from Insert mode.

5. Saving and Quitting:

‘:w’: Save changes.

‘:q’: Quit the editor.

‘:wq’: Save and quit.

‘:q!’: Quit without saving (force quit).

6. Search and Replace in Command-line Mode:

‘:/pattern’: Search for a specific pattern.

‘:%s/pattern/replace/g’: Replace all occurrences of a pattern with another string.

These are just some of the basic commands in vi. The editor offers many more features and commands that can be useful for both simple text editing and more complex tasks. Remember that mastering vi might take some practice, but once you become proficient, it can be a powerful tool for working with text files in a command-line environment.

Vi Commands

List of common vi commands categorized by the modes in which they are used:

Normal Mode Commands:

Navigation:

‘h’: Move cursor left.

‘j’: Move cursor down.

‘k’: Move cursor up.

‘l’: Move cursor right.

‘w’: Move to the beginning of the next word.

‘b’: Move to the beginning of the previous word.

‘0’: Move to the beginning of the current line.

‘$’: Move to the end of the current line.

‘gg’: Move to the beginning of the file.

‘G’: Move to the end of the file.

‘:{line_number}’: Move to a specific line number.

‘%’: Move to the matching parenthesis or brace.

Editing:

‘x’: Delete the character under the cursor.

‘dd’: Delete the current line.

‘yy’: Copy the current line.

‘p’: Paste the copied or deleted content after the cursor.

‘P’: Paste before the cursor.

‘u’: Undo the last action.

‘Ctrl + r’: Redo.

‘.’: Repeat the last change.

Searching:

‘/pattern’: Search for a pattern forward.

‘?pattern’: Search for a pattern backward.

‘n’: Move to the next search result.

‘N’: Move to the previous search result.

Marks and Movement:

‘ma’: Set mark 'a' at the current cursor position.

‘'a’: Jump to the mark 'a'.

‘`a’: Jump to the beginning of the line containing mark 'a'.

‘'’: Jump back to the position before the latest jump.

‘Ctrl’ + ‘o’: Jump to older position.

‘Ctrl’ + ‘i’: Jump to newer position.

Copy and Paste Between Files:

‘:e {filename}’: Open a new file for editing.

‘:vsp {filename}’: Open a new file in a vertical split.

‘:sp {filename}’: Open a new file in a horizontal split.

‘:bnext’ or ‘:bn’: Move to the next buffer.

‘:bprev’ or ‘:bp’: Move to the previous buffer.

‘:bd’: Close the current buffer (close the file).

‘:ls’: List all open buffers.

‘:b {buffer_number}’: Switch to a specific buffer.

Insert Mode Commands:

❖ **Inserting Text:**

Press ‘i’ to enter Insert mode before the cursor.

Press ‘I’ to enter Insert mode at the beginning of the line.

Press ‘a’ to enter Insert mode after the cursor.

Press ‘A’ to enter Insert mode at the end of the line.

Press ‘o’ to open a new line below and enter Insert mode.

Press ‘O’ to open a new line above and enter Insert mode.

Press ‘Esc’ to return to Normal mode.

Command-Line Mode Commands:

- ✓ **Saving and Quitting:**

  ‘:w’: Save changes.

  ‘:q’: Quit the editor.

  ‘:wq’ or ‘:x’: Save and quit.

  ‘:q!’: Quit without saving (force quit).

  ‘:wqa’: Save changes in all open windows and quit.

- ✓ **Search and Replace:**

‘:%s/pattern/replace/g’: Replace all occurrences of a pattern with another string.

‘:s/pattern/replace/’: Replace the first occurrence of a pattern in the current line.

‘:s/pattern/replace/g’: Replace all occurrences of a pattern in the current line.

- ✓ **Moving Between Files:**

‘:e {filename}’: Edit a different file.

‘:sp {filename}’: Open a file in a horizontal split.

‘:vsp {filename}’: Open a file in a vertical split.

‘:b {buffer_number}’: Switch to a specific buffer.

These are just some of the most common vi commands. Remember that vi has a wide range of features and commands that can be quite powerful once you become comfortable with the editor's workflow.