Linux Shell Scripting:

Unit-I

Basic bash Shell Commands: Interacting with the shell - Traversing the file system - Listing files and directories - Managing files and directories - Viewing file contents. Basic Script Building: Using multiple commands Creating a script file Displaying messages - Using variables Redirecting input and output - Pipes - Performing math - Exiting the script. 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.

Unit-II

More Structured Commands: Looping with for statement Iterating with the until statement Using the while statement - Combining loops - Redirecting loop output. Handling User Input: Passing parameters - Tracking parameters - Being shifty - Working with options -Standardizing options - Getting user input. Script Control: Handling signals - Running scripts in the background Forbidding hang-ups - Controlling a Job Modifying script priority -Automating script execution.

Unit-III

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. Writing Scripts for Graphical Desktops: Creating text menus - Building text window widgets - Adding X Window graphics. Introducing sed and gawk: Learning about the sed Editor - Getting introduced to the gawk Editor - Exploring sed Editor basics**.**

Unit-IV

Regular Expressions: Defining regular expressions - Looking at the basics - Extending our patterns - Creating expressions. 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. Advanced gawk: Reexamining gawk - Using variables in gawk - Using structured commands - Formatting the printing - Working with functions.

Unit-V

Working with Alternative Shells: Understanding the dash shell - Programming in the dash shell- Introducing the zsh shell - Writing scripts for zsh. Writing Simple Script Utilities: Automating backups - Managing user accounts - Watching disk space. 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 1

Basic Shell Commands:

1. What is Bash?

Bash (Bourne Again Shell) is a command-line interpreter that provides a user interface for interacting with the operating system. It is commonly used in Unix-based systems like Linux and macOS.

2. Basic Shell Commands:

Below are some fundamental commands used in Bash:

a. Navigating the File System:

- pwd: Print the current working directory.
- ls: List the contents of a directory.
  - ls -l: Display detailed information about files and directories.
  - ls -a: List hidden files.
- cd <directory>: Change to another directory.
  - cd ..: Move up one level in the directory structure.
  - cd ~: Move to the home directory.

b. File Operations:

- touch <filename>: Create an empty file.
- cat <filename>: Display the content of a file.
- cp <source> <destination>: Copy a file or directory.
- mv <source> <destination>: Move or rename a file or directory.
- rm <filename>: Remove (delete) a file.
  - rm -r <directory>: Remove a directory and its contents.

c. Viewing and Editing Files:

- less <filename>: View the content of a file one page at a time.
- nano <filename>: Open a simple text editor to modify files.
- head <filename>: Display the first 10 lines of a file.

- tail <filename>: Display the last 10 lines of a file.
    - tail -f <filename>: Monitor the file for real-time updates (commonly used for log files).

d. System Information:

- whoami: Show the current logged-in username.
- uname -a: Show system and kernel information.
- df -h: Display disk usage of file systems.
- top: Display a real-time view of running processes and system resource usage.

e. Directory and File Permissions:

- chmod <permissions> <file>: Change file permissions.
    - Example: chmod 755 file.txt (grants read, write, execute to owner, read and execute to others).
- chown <user>:<group> <file>: Change the ownership of a file or directory.

f. Process Management:

- ps: List running processes.
- kill <PID>: Terminate a process by its Process ID (PID).
- killall <process_name>: Terminate all processes with a specific name.
- bg: Resume a suspended job in the background.
- fg: Bring a background job to the foreground.

g. Searching and Finding Files:

- find <directory> -name <filename>: Search for a file in a directory.
- grep <pattern> <file>: Search for a specific pattern in a file.
    - Example: grep "error" logfile.txt (searches for the word "error" in a log file).

h. Package Management (Linux):

- apt-get update: Update the list of available packages (Debian/Ubuntu).
- apt-get install <package>: Install a software package.
- yum install <package>: Install a software package (CentOS/RHEL).

3. Shell Redirection and Piping:

- >: Redirect output to a file (overwrites existing content).
    - Example: ls > filelist.txt
- >>: Append output to a file (preserves existing content).
    - Example: echo "text" >> file.txt
- <: Use a file as input for a command.
    - Example: sort < file.txt
- |: Pipe output from one command as input to another.
    - Example: ls | grep "pattern"

4. Shell Scripting:

- Create a Bash script file by using nano script.sh.
- Add the #!/bin/bash shebang at the top of the file.
- Write a series of Bash commands inside the script.
- Make the script executable using chmod +x script.sh.
- Execute the script with ./script.sh.

5. Wildcards and Globbing:

- *: Matches any number of characters.
    - Example: ls *.txt (lists all .txt files).
- ?: Matches a single character.
    - Example: ls file?.txt (matches file1.txt, file2.txt, etc.).

6. Aliases:

- You can create custom shortcuts for long commands using alias.
    - Example: alias ll='ls -la'
- To make it permanent, add the alias to your .bashrc or .bash_profile file.

Traversing the file system:

Traversing the file system in Bash involves moving through directories and viewing their contents. Below are common commands for file system navigation:

1. View Current Location:

- **pwd** (Print Working Directory):
    - Displays the full path of your current location in the file system.

Example:

```
bash
$ pwd
```

/home/user/Documents

2. Listing Files and Directories:

- **ls**:
    - Lists files and directories in the current directory.

Example:

```
bash
$ ls
```

file1.txt file2.txt directory1 directory2

- **Options for ls**:
    - ls -l: Displays detailed information (file size, permissions, owner, etc.).
    - ls -a: Lists all files, including hidden files (those starting with .).
    - ls -lh: Lists files in human-readable format (shows file sizes in KB, MB).

3. Changing Directories:

- **cd** (Change Directory):
    - Moves between directories.

Example:

```
bash
$ cd /home/user/Documents
```

- To go back to the previous directory:

```
bash
$ cd ..
```

- **Common cd usage**:
    - cd ~: Moves to the home directory.
    - cd /: Moves to the root directory.
    - cd -: Switches to the previous working directory.

4. Absolute and Relative Paths:

- **Absolute Path**:
    - Starts from the root directory / and provides the full path.

Example:

```
bash
$ cd /home/user/Documents/Project
```

- **Relative Path**:
    - Starts from your current directory and navigates without needing the full path.

Example (if you are in /home/user/):

```
bash
$ cd Documents/Project
```

5. Viewing Directory Structure:

- **tree**:
    - Displays the directory structure in a tree format (if installed).

Example:

```
bash
$ tree
```

- To install tree on Ubuntu or Debian:

```
bash
$ sudo apt-get install tree
```

6. Creating and Navigating into Directories:

- **mkdir <directory_name>**:
    - Creates a new directory.
    - Example:

```
bash
$ mkdir new_directory
```

- **cd new_directory**:
    - Move into the newly created directory.
    - Example:

```
bash
$ cd new_directory
```

7. Viewing Hidden Files:

- **Hidden Files**:
    - Files that start with a dot (.) are hidden.
    - Example:

```
bash
$ ls -a
.bashrc  .profile  .hiddenfile  file1.txt
```

8. Finding Files and Directories:

- **find**:
    - Searches for files or directories in a given location.
    - Example (search for a file named file.txt):

```
bash
$ find /home/user -name file.txt
```

- **locate** (alternative search command):
    - Searches files quickly by using an updated database of your file system.
    - Example:

```
bash
$ locate file.txt
```

9. Using Wildcards to Traverse:

- **Wildcards**:
    - Wildcards allow flexible file and directory selection.
    - *: Matches any number of characters.
        - Example:

```
bash
$ ls *.txt
```

Lists all .txt files in the directory.

- ?: Matches a single character.
    - Example:

```
bash
$ ls file?.txt
```

Matches files like file1.txt and file2.txt.

10. Navigating Directories Efficiently:

- **pushd and popd**:
    - pushd <directory>: Moves to a new directory and pushes the current one onto a stack.
    - popd: Pops the last directory from the stack and moves to it.
    - Example:

```
bash
$ pushd /var/log
$ popd
```

Listing files and directories:

In Bash, listing files and directories is a fundamental task. Below are various commands and options you can use to list files and directories effectively:

Basic Listing:

- **ls**:
    - Lists files and directories in the current working directory.
    - Example:

Bash Code

```
$ ls
file1.txt  file2.txt  directory1  directory2
```

Listing Files with Details:

- **ls -l**:
    - Provides detailed information about each file and directory, such as permissions, owner, file size, and modification date.
    - Example:

Bash Code

```
$ ls -l
total 8
-rw-r--r-- 1 user user  1024 Sep 27 14:50 file1.txt
drwxr-xr-x 2 user user  4096 Sep 27 14:55 directory1
```

Including Hidden Files:

- **ls -a**:
    - Lists all files, including hidden files (files and directories that start with a dot .).
    - Example:

Bash Code

```
$ ls -a
.  ..  .bashrc  .profile  file1.txt
```

Combining Detailed and Hidden Files:

- **ls -la**:
    - Lists detailed information for all files, including hidden ones.
    - Example:

bash

Code

```
$ ls -la
total 16
drwxr-xr-x 2 user user  4096 Sep 27 14:55 .
drwxr-xr-x 4 user user  4096 Sep 27 14:50 ..
-rw-r--r-- 1 user user  1024 Sep 27 14:50 .bashrc
-rw-r--r-- 1 user user   512 Sep 27 14:50 file1.txt
```

Human-Readable File Sizes:

- **ls -lh**:
    - Lists files with sizes in a human-readable format (e.g., KB, MB).
    - Example:

Bash Code

```
$ ls -lh
total 8.0K
-rw-r--r-- 1 user user 1.0K Sep 27 14:50 file1.txt
drwxr-xr-x 2 user user 4.0K Sep 27 14:55 directory1
```

Listing by Modification Time:

- **ls -lt**:
    - Lists files sorted by modification time, with the most recently modified files appearing first.
    - Example:

Bash Code

```
$ ls -lt
-rw-r--r-- 1 user user 1024 Sep 27 14:50 file1.txt
drwxr-xr-x 2 user user 4096 Sep 26 13:45 directory1
```

Recursive Listing:

- **ls -R**:
    - Lists all files and directories recursively, showing the contents of subdirectories as well.
    - Example:

Bash Code

```
$ ls -R

.:

file1.txt  directory1


./directory1:

file2.txt
```

Sorting Files by Size:

- **ls -lS**:
    - Lists files sorted by size, largest to smallest.
    - Example:

Bash Code

```
$ ls -lS
-rw-r--r-- 1 user user 2048 Sep 27 14:50 large_file.txt
-rw-r--r-- 1 user user 1024 Sep 27 14:50 file1.txt
```

Listing with Inode Numbers:

- **ls -i**:
    - Lists files with their inode numbers (unique identifier for each file).
    - Example:

Bash Code

```
$ ls -i
128849023 file1.txt  128849024 directory1
```

Sorting Files by Extension:

- **ls -X**:
    - Lists files sorted by extension.
    - Example:

Bash Code

```
$ ls -X
script.sh  file1.txt  notes.md
```

Traversing file system , listing files and direcotries:

. Traversing the File System

Traversing the file system in Bash involves moving between directories to navigate the directory hierarchy.

- **`pwd` (Print Working Directory)**:
  Displays the absolute path of the current directory you're working in.

  ```
  bash

  $ pwd
  /home/user/projects
  ```

- **`cd` (Change Directory)**:
  Changes your current working directory.
    - **Move to a specific directory**:

      ```
      bash

      $ cd /path/to/directory
      ```

    - **Move to the parent directory**:

      ```
      bash

      $ cd ..
      ```

    - **Move to your home directory**:

      ```
      bash

      $ cd
      ```

    - **Move to the previous directory**:

      ```
      bash

      $ cd -
      ```

- **Relative Path vs. Absolute Path**:
    - **Relative path**: Moves relative to the current directory.

      ```
      bash

      $ cd Documents  # If you are in /home/user, moves to
      /home/user/Documents
      ```

- **Absolute path**: Full path starting from root `/`.

  ```bash
  $ cd /home/user/Documents
  ```

- **`tree` Command** (if installed):
  Shows a visual tree-like structure of the directory and its subdirectories.

  ```bash
  $ tree
  ```

---

2. Listing Files and Directories

Listing files and directories in Bash is achieved using the `ls` command. This command has various options to modify its output.

- **`ls`**:
  Lists files and directories in the current directory.

  ```bash
  $ ls
  file1.txt  directory1  file2.txt
  ```

- **`ls -l`**:
  Lists files with detailed information, such as permissions, owner, size, and modification date.

  ```bash
  $ ls -l
  total 16
  -rw-r--r-- 1 user user 1024 Sep 27 14:50 file1.txt
  drwxr-xr-x 2 user user 4096 Sep 27 14:55 directory1
  ```

- **`ls -a`**:
  Lists all files, including hidden files (files starting with `.`).

  ```bash
  $ ls -a
  .  ..  .bashrc  .profile  file1.txt
  ```

- **`ls -lh`**:
  Lists files with sizes in human-readable format (KB, MB, GB).

  ```bash
  $ ls -lh
  total 8.0K
  -rw-r--r-- 1 user user 1.0K Sep 27 14:50 file1.txt
  drwxr-xr-x 2 user user 4.0K Sep 27 14:55 directory1
  ```

- **`ls -lt`**:
  Lists files sorted by modification time, with the most recently modified files first.

  ```bash
  $ ls -lt
  -rw-r--r-- 1 user user 1024 Sep 27 14:50 file1.txt
  ```

- **`ls -R`**:
  Recursively lists all files and directories, including the contents of subdirectories.

  ```bash
  $ ls -R
  .:
  file1.txt  directory1

  ./directory1:
  file2.txt
  ```

- **`ls -lS`**:
  Lists files sorted by size, with the largest files first.

  ```bash
  $ ls -lS
  -rw-r--r-- 1 user user 2048 large_file.txt
  -rw-r--r-- 1 user user 1024 file1.txt
  ```

- **`ls -X`**:
  Lists files sorted by extension.

  ```bash
  $ ls -X
  script.sh  file1.txt  notes.md
  ```

- **`ls -i`**:
  Lists files along with their inode numbers.

  ```bash
  $ ls -i
  128849023 file1.txt  128849024 directory1
  ```

---

3. Copying Files and Directories

The `cp` command in Bash allows you to copy files and directories.

- **Basic Syntax**:

```
bash

cp [options] <source> <destination>
```

- **Copying a Single File**:
  - To copy a file from one location to another.

```
bash

$ cp file1.txt /path/to/destination/
```

- **Copying Multiple Files**:
  - To copy multiple files to a directory.

```
bash

$ cp file1.txt file2.txt /path/to/destination/
```

- **Copying Directories**:
  - Use the `-r` option to copy directories and their contents recursively.

```
bash

$ cp -r directory1 /path/to/destination/
```

- **`cp` Options**:
  - `-i` (interactive): Prompts before overwriting existing files.
  - `-v` (verbose): Shows the files being copied.
  - `-u` (update): Only copies if the source file is newer than the destination file.

Summary of Common Commands:

Command	Description
`pwd`	Shows the current directory
`cd <path>`	Changes to the specified directory
`cd ..`	Moves up one directory
`ls`	Lists files and directories
`ls -l`	Lists files with detailed information
`ls -a`	Lists all files, including hidden ones
`ls -lh`	Lists files with human-readable sizes
`cp <source> <destination>`	Copies files or directories
`cp -r <source_dir> <destination>`	Copies directories recursively

. Traversing the File System

Traversing the file system in Bash involves moving between directories to navigate the directory hierarchy.

- **pwd (Print Working Directory)**:
  Displays the absolute path of the current directory you're working in.

```
bash


$ pwd

/home/user/projects
```

- **cd (Change Directory)**:
  Changes your current working directory.
  - **Move to a specific directory**:

```
bash


$ cd /path/to/directory
```

  - **Move to the parent directory**:

```
bash


$ cd ..
```

  - **Move to your home directory**:

```
bash


$ cd
```

- **Move to the previous directory**:

bash

```
$ cd -
```

- **Relative Path vs. Absolute Path**:
  - **Relative path**: Moves relative to the current directory.

bash

```
$ cd Documents  # If you are in /home/user, moves to /home/user/Documents
```

  - **Absolute path**: Full path starting from root /.

bash

```
$ cd /home/user/Documents
```

- **tree Command** (if installed):
  Shows a visual tree-like structure of the directory and its subdirectories.

bash

```
$ tree
```

---

2. Listing Files and Directories

Listing files and directories in Bash is achieved using the ls command. This command has various options to modify its output.

- **ls**:
  Lists files and directories in the current directory.

bash

```
$ ls
file1.txt  directory1  file2.txt
```

- **ls -l**:
  Lists files with detailed information, such as permissions, owner, size, and modification date.

bash

```
$ ls -l
total 16
-rw-r--r-- 1 user user 1024 Sep 27 14:50 file1.txt
drwxr-xr-x 2 user user 4096 Sep 27 14:55 directory1
```

- **ls -a**:
  Lists all files, including hidden files (files starting with .).

bash

```
$ ls -a
. .. .bashrc  .profile  file1.txt
```

- **ls -lh**:
  Lists files with sizes in human-readable format (KB, MB, GB).

bash

```
$ ls -lh
total 8.0K
-rw-r--r-- 1 user user 1.0K Sep 27 14:50 file1.txt
drwxr-xr-x 2 user user 4.0K Sep 27 14:55 directory1
```

- **ls -lt**:
  Lists files sorted by modification time, with the most recently modified files first.

bash

```
$ ls -lt
-rw-r--r-- 1 user user 1024 Sep 27 14:50 file1.txt
```

- **ls -R**:
  Recursively lists all files and directories, including the contents of subdirectories.

bash

```
$ ls -R
.:
file1.txt  directory1

./directory1:
file2.txt
```

- **ls -lS**:
  Lists files sorted by size, with the largest files first.

bash

```
$ ls -lS
-rw-r--r-- 1 user user 2048 large_file.txt
-rw-r--r-- 1 user user 1024 file1.txt
```

- **ls -X**:
  Lists files sorted by extension.

bash

```
$ ls -X
script.sh  file1.txt  notes.md
```

- **ls -i**:
  Lists files along with their inode numbers.

bash

```
$ ls -i
128849023 file1.txt  128849024 directory1
```

---

3. Copying Files and Directories

The cp command in Bash allows you to copy files and directories.

- **Basic Syntax**:

bash

```
cp [options] <source> <destination>
```

- **Copying a Single File**:
  - To copy a file from one location to another.

bash

```
$ cp file1.txt /path/to/destination/
```

- **Copying Multiple Files**:
  - To copy multiple files to a directory.

bash

```
$ cp file1.txt file2.txt /path/to/destination/
```

- **Copying Directories**:
  - Use the -r option to copy directories and their contents recursively.

bash

```
$ cp -r directory1 /path/to/destination/
```

- **cp Options**:
  - -i (interactive): Prompts before overwriting existing files.
  - -v (verbose): Shows the files being copied.
  - -u (update): Only copies if the source file is newer than the destination file.

Summary of Common Commands:

Managing files and **directories**:

Managing files and directories in Bash involves tasks like creating, renaming, moving, and deleting files or directories. Here's how to perform these tasks effectively:

Command	Description
21	
pwd	Shows the current directory
cd <path>	Changes to the specified directory
cd ..	Moves up one directory
ls	Lists files and directories
ls -l	Lists files with detailed information
ls -a	Lists all files, including hidden ones
ls -lh	Lists files with human-readable sizes
cp <source> <destination>	Copies files or directories
cp -r <source_dir> <destination>	Copies directories recursively

1. Managing Files:

Creating Files:

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

  bash

  ```
  $ touch filename.txt
  ```

- **cat >**:
  Creates a new file and allows immediate text input. Press `Ctrl+D` to save and exit.

  bash

  ```
  $ cat > filename.txt
  This is a new file.
  ```

Renaming Files:

- **mv (move)**:
  Renames a file by moving it to a new name.

  bash

  ```
  $ mv oldname.txt newname.txt
  ```

Moving Files:

- **mv**:
  Moves a file to a different directory.

  bash

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

Deleting Files:

- **rm**:
  Removes (deletes) a file. **Warning**: Once deleted, files are typically not recoverable.

  bash

  ```
  $ rm filename.txt
  ```

- **rm -i**:
  Asks for confirmation before deleting a file.

  bash

  ```
  $ rm -i filename.txt
  ```

---

2. Managing Directories:

Creating Directories:

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

  bash

  ```
  $ mkdir new_directory
  ```

- **mkdir -p**:
  Creates nested directories in one command.

  bash

  ```
  $ mkdir -p /path/to/new_directory/sub_directory
  ```

Renaming Directories:

- **mv**:
  Renames a directory.

  bash

  ```
  $ mv old_directory new_directory
  ```

Moving Directories:

- **mv**:
  Moves a directory to a new location.

```bash
$ mv directory_name /path/to/new_location/
```

Deleting Directories:

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

  ```bash
  $ rmdir empty_directory
  ```

- **rm -r**:
  Removes a directory and all its contents (recursively).

  ```bash
  $ rm -r directory_name
  ```

- **rm -rf**:
  Force removes a directory and its contents without confirmation.

  ```bash
  $ rm -rf directory_name
  ```

3. Viewing File Contents:

When you want to view or read the contents of a file without modifying it, several commands can be used in Bash.

cat:

- Concatenates and displays the content of a file.

  ```bash
  $ cat filename.txt
  ```

less:

- Allows you to scroll through the content of a file one screen at a time.
  Navigation: Use the arrow keys to scroll, press q to exit.

  ```bash
  $ less filename.txt
  ```

more:

- Similar to less, but only allows forward movement in the file.

  ```bash
  $ more filename.txt
  ```

head:

- Displays the first 10 lines of a file by default.

  ```bash
  $ head filename.txt
  ```

- Display a specific number of lines:

  ```bash
  $ head -n 20 filename.txt  # Shows first 20 lines
  ```

tail:

- Displays the last 10 lines of a file by default.

  ```bash
  $ tail filename.txt
  ```

- Display a specific number of lines from the end:

  ```bash
  $ tail -n 20 filename.txt  # Shows last 20 lines
  ```

- Monitor file updates in real-time (useful for log files):

  ```bash
  $ tail -f filename.log
  ```

nl:

- Displays the contents of a file with line numbers.

  ```bash
  $ nl filename.txt
  ```

grep:

- Searches for a specific pattern in a file and displays matching lines.

```
bash

$ grep "search_term" filename.txt
```

Summary of Commands for Managing Files and Directories:

Command	Description
touch filename	Creates a new empty file
mv oldname newname	Renames or moves a file/directory
rm filename	Deletes a file
mkdir directory_name	Creates a new directory
rmdir directory_name	Removes an empty directory
rm -r directory_name	Deletes a directory and all its contents

Summary of Commands for Viewing File Content:

Command	Description
cat filename	Displays the entire content of a file
less filename	Scrolls through the content one screen at a time
head filename	Displays the first 10 lines of a file
tail filename	Displays the last 10 lines of a file
grep "pattern" filename	Searches for a pattern in a file and displays matches

These commands will help you manage and view files efficiently when working in a Linux/Bash environment.

UNIT 2

1. Looping with the for Statement

The for loop in shell scripting is used to iterate over a list of items, making it ideal for performing repetitive tasks. The syntax and behavior may vary slightly between different shells, but the basic structure is generally consistent.

Syntax:

```
bash

for variable in list
do
  command1
  command2
  ...
done
```

Example:

```
bash

for i in 1 2 3 4 5
do
  echo "Iteration $i"
done
```

Explanation:

- for i in 1 2 3 4 5: Defines the loop and assigns each value (1 through 5) to the variable i on each iteration.
- echo "Iteration $i": Outputs the current value of i.
- done: Marks the end of the loop.

The for loop can also iterate through files, command outputs, or a range of numbers using {} syntax:

```
bash

for i in {1..5}
do
  echo "Iteration $i"
done
```

2. Iterating with the until Statement

The until loop is another looping construct, similar to the while loop, but it keeps iterating until a specified condition becomes true (i.e., it runs while the condition is false).

Syntax:

```
bash

until [ condition ]
do
  command1
  command2
  ...
done
```

Example:

```
bash

counter=1
until [ $counter -gt 5 ]
do
  echo "Counter is $counter"
  ((counter++))
done
```

Explanation:

- until [$counter -gt 5]: The loop continues until counter is greater than 5.
- echo "Counter is $counter": Prints the value of counter.
- ((counter++)): Increments the counter variable by 1.
- done: Ends the loop.

This loop is useful when you want to execute a block of code until a certain condition is met.

3. Using the while Statement

The while loop continues to run as long as a specified condition is true. It is one of the most commonly used loops in shell scripting.

Syntax:

```
bash
```

```
while [ condition ]
do
  command1
  command2
  ...
done
```

Example:

```
bash

counter=1
while [ $counter -le 5 ]
do
  echo "Counter is $counter"
  ((counter++))
done
```

Explanation:

- while [$counter -le 5]: Runs as long as counter is less than or equal to 5.
- ((counter++)): Increments the counter.
- This structure is effective for situations where the number of iterations is determined by a condition rather than a set number of elements.

4. Combining Loops

You can nest loops within each other (i.e., place a loop inside another loop) to perform more complex iterations, such as iterating through multi-dimensional arrays or performing calculations.

Example:

```
bash

for i in 1 2 3
do
  for j in a b c
  do
    echo "Outer loop: $i, Inner loop: $j"
  done
done
```

Explanation:

- The outer for loop iterates through the numbers 1, 2, and 3.
- The inner for loop iterates through the letters a, b, and c for each iteration of the outer loop.
- This combination of loops prints all combinations of i and j.

5. Redirecting Loop Output

Redirecting output in loops allows you to control where the output goes, which can be useful when you need to save loop results to a file or handle errors.

Redirecting Loop Output to a File

To redirect all output from a loop to a file, you can use the redirection operator (> or >>) after done.

Example:

```
bash

for i in {1..5}
do
   echo "This is line $i"
done > output.txt
```

Explanation:

- > output.txt: Redirects the output of the loop to output.txt, overwriting any existing content.
- To append to a file instead, use >> output.txt.

Redirecting Error Output

To capture only errors, use 2> for standard error redirection.

Example:

```
bash

for i in {1..5}
do
   echo "Processing $i"
   ls /nonexistentdir 2>> errors.log
done
```

Explanation:

- 2>> errors.log: Appends any errors generated by ls /nonexistentdir to errors.log.

Combining Standard Output and Error

You can redirect both standard output and standard error to a file using &> (in Bash).

Example:

```
bash

for i in {1..5}
do
   echo "Processing $i"
   ls /nonexistentdir
done &> combined_output.log
```

Explanation:

- &> combined_output.log: Redirects both standard output and error output to combined_output.log.\

1. Passing Parameters

In shell scripting, you can pass parameters to a script by typing them after the script name. These parameters are accessed within the script using positional parameters like $1, $2, etc., where $1 is the first parameter, $2 is the second, and so on.

Example:

```
bash

#!/bin/bash

echo "First parameter: $1"

echo "Second parameter: $2"
```

Usage:

```
bash

./script.sh Hello World
```

Output:

```
sql
First parameter: Hello
Second parameter: World
```

Special Variables:

- $0: The name of the script.
- $#: The number of positional parameters.
- $@: All parameters passed to the script as a list.

2. Tracking Parameters

Tracking parameters means checking the values of parameters passed to the script and using them as conditions. This helps ensure that the user has provided the required parameters and can also be used to set default values if parameters are missing.

Example:

```
bash
#!/bin/bash
if [ -z "$1" ]; then
  echo "No parameters provided."
  exit 1
fi
echo "Running script with parameter: $1"
```

Explanation:

- [-z "$1"]: Checks if $1 (the first parameter) is empty. If so, it notifies the user and exits.

3. "Being Shifty": Using the shift Command

The shift command shifts all positional parameters one place to the left, discarding the first parameter and making $2 the new $1, $3 the new $2, and so on. This is useful when working with an unknown number of arguments.

Example:

```
bash
#!/bin/bash
while [ $# -gt 0 ]; do
  echo "Parameter: $1"
  shift
done
```

Explanation:

- while [$# -gt 0]: Loops as long as there are parameters left.
- shift: Discards the first parameter and reassigns the rest.

Usage:

```
bash
./script.sh one two three
```

Output:

```
sql
Parameter: one
Parameter: two
Parameter: three
```

4. Working with Options (Using Flags)

Options or flags, often prefixed with - (e.g., -a), allow users to enable certain behaviors within a script. The getopts command helps parse these options easily.

Example with getopts:

```
bash
```

```
#!/bin/bash
while getopts ":ab:c:" option; do
  case $option in
    a) echo "Option -a enabled";;
    b) echo "Option -b with argument: $OPTARG";;
    c) echo "Option -c with argument: $OPTARG";;
    *) echo "Invalid option";;
  esac
done
```

Explanation:

- getopts ":ab:c:": The options are -a, -b, and -c. The : after b and c indicates that these options require arguments.
- $OPTARG: Captures the argument value for options requiring one.

Usage:

```
bash
./script.sh -a -b argument_for_b -c argument_for_c
```

Output:

```
vbnet
Option -a enabled
Option -b with argument: argument_for_b
Option -c with argument: argument_for_c
```

5. Standardizing Options

Standardizing options means creating consistent and predictable ways for users to interact with a script, typically by following conventions for common options like -h for help, -v for version, or -o for output. This makes scripts easier to understand and use.

Example:

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

```
while getopts ":hvo:" option; do
  case $option in
    h) echo "Usage: script.sh [-h] [-v] [-o output_file]"; exit 0;;
    v) echo "Script version 1.0"; exit 0;;
    o) output_file=$OPTARG;;
    *) echo "Invalid option"; exit 1;;
  esac
done
```

Explanation:

- -h: Displays a usage message.
- -v: Prints the script version.
- -o: Allows the user to specify an output file.

Usage:

```
bash
./script.sh -h
./script.sh -v
./script.sh -o filename.txt
```

6. Getting User Input (Interactive Input)

Interactive input prompts users to enter information during script execution using the read command. This approach is useful for gathering sensitive information like passwords or when specific details are needed.

Example:

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

Explanation:

- read name: Waits for the user to input a value and assigns it to the variable name.

Usage: When run, the script will prompt the user for their name and then greet them.

Prompting for Hidden Input (e.g., passwords):

```
bash
#!/bin/bash
read -sp "Enter your password: " password
echo -e "\nPassword received."
```

Explanation:

- -s: Hides the input text, useful for password entry.

1. Handling Signals

Signals are notifications sent to a process to trigger specific actions. Common signals include SIGINT (interrupt), SIGTERM (terminate), and SIGHUP (hangup). Using the trap command, you can specify how your script should respond to these signals.

Basic Syntax for trap:

```
bash
trap 'commands' SIGNAL
```

Example:

```
bash
#!/bin/bash
trap 'echo "Caught SIGINT signal"; exit' SIGINT
trap 'echo "Caught SIGTERM signal"; exit' SIGTERM
echo "Script is running. Press Ctrl+C to interrupt."
while true; do
  sleep 1
done
```

Explanation:

- trap 'commands' SIGNAL: Executes specified commands when the signal is received.
- SIGINT: Typically generated by pressing Ctrl+C. This trap will catch it and execute the echo message and exit.

You can trap multiple signals or customize how a script handles unexpected exits using signals.

2. Running Scripts in the Background

Running a script in the background allows you to continue working in the terminal while the script executes. This is done by appending & at the end of the command.

Example:

```
bash
./long_running_script.sh &
```

Explanation:

- The & character places the script in the background.
- You can check background jobs by running jobs and see active background tasks.

To bring a background job to the foreground, use:

```
bash
fg %job_number
```

You can get the job_number by running jobs.

3. Forbidding Hang-ups (nohup)

When running a script over SSH or in any environment where the terminal might close, using nohup (no hang-up) prevents the script from stopping if the terminal is disconnected.

Example:

```
bash
nohup ./long_running_script.sh &
```

Explanation:

- nohup: Allows the script to keep running even if the terminal session ends.
- By default, nohup redirects output to nohup.out unless specified otherwise.

4. Controlling a Job

Controlling a job in Linux involves pausing, resuming, or stopping the execution of a background or foreground job. This is often done with job control commands.

Commands for Job Control:

- jobs: Lists current jobs.
- bg %job_number: Resumes a suspended job in the background.
- fg %job_number: Brings a background job to the foreground.
- kill %job_number: Terminates a job.

Example:

```
bash
./long_running_script.sh &
sleep 5
jobs
kill %1
```

Explanation:

- This sequence starts a script in the background, lists running jobs, and then terminates the first job.

5. Modifying Script Priority (nice and renice)

In Linux, you can set the priority of a script using nice when starting it, or change its priority during execution with renice. Lower priority numbers mean higher priority for CPU time, and higher numbers indicate lower priority.

Setting Priority with nice:

```
bash
nice -n 10 ./script.sh
```

Explanation:

- -n 10: Sets the priority of the script to 10. The default is 0, and typical values range from -20 (highest priority) to 19 (lowest).

Modifying Priority of a Running Script with renice:

```
bash
renice +5 -p 1234
```

Explanation:

- +5: Changes the priority to 5 (lowering priority).
- -p 1234: Specifies the process ID (PID) of the running script.

6. Automating Script Execution (Using cron)

The cron daemon is used to schedule commands to run at specific times or intervals, making it ideal for automating script execution. Cron jobs are configured in a file called the crontab.

Syntax for Crontab Entries:

```
plaintext
* * * * * command_to_run
```

- The five * symbols represent the minute, hour, day of the month, month, and day of the week, respectively.

Example Crontab Entry: To run a script every day at midnight:

```
bash
0 0 * * * /path/to/script.sh
```

Explanation:

- 0 0 * * *: Schedules the script to run at 00:00 (midnight) every day.
- /path/to/script.sh: Specifies the path to the script to execute.

Managing Cron Jobs:

- To edit your cron jobs, run crontab -e.
- To list scheduled cron jobs, run crontab -l.

Alternatively, for one-time scheduling, at can also be used to run a command at a specified time.

UNIT 3

Basic Script Functions

A function is a block of code that performs a specific task. In shell scripting, functions help in organizing code into modular blocks, allowing for easier maintenance and reuse.

Defining a Function

Functions in shell scripts are defined using the following syntax:

```
sh
function_name() {
    # Commands
}
```

Example: Basic Function

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

In this example, greet is a function that takes one argument and prints a greeting message.

Returning a Value

Returning a Status Code

Functions in shell scripting typically return a status code (an integer value). The return statement is used to return a status code.

```
sh
#!/bin/bash
check_file() {
    if [ -f "$1" ]; then
        return 0  # Success
    else
```

```
        return 1  # Failure
    fi
}
check_file "test.txt"
if [ $? -eq 0 ]; then
    echo "File exists."
else
    echo "File does not exist."
fi
```

In this example, the function check_file returns 0 if the file exists and 1 if it doesn't. The status code is checked using the special variable $?.

Returning a Value via echo

Since functions cannot return non-numeric values directly, you can use echo to return a string or other data.

```
sh
#!/bin/bash
get_date() {
    echo $(date)
}
current_date=$(get_date)
echo "Current date and time: $current_date"
```

Here, get_date function echoes the current date and time, which is captured and stored in the variable current_date.

Using Variables in Functions

Scope of Variables

Variables in shell functions can be global or local. By default, variables are global, meaning they can be accessed anywhere in the script. You can use the local keyword to restrict the scope of a variable to within the function.

Example: Global Variables

```
sh
#!/bin/bash
count=0
increment() {
    count=$((count + 1))
}
increment
echo "Count: $count"
```

In this example, the variable count is global and is modified within the increment function.

Example: Local Variables

```
sh
#!/bin/bash
increment() {
    local count=0
    count=$((count + 1))
    echo "Count inside function: $count"
}
increment
echo "Count outside function: $count"  # This will be empty
```

Here, the variable count inside the function increment is local and does not affect the global scope.

Array and Variable Functions

Using Arrays in Functions

Arrays can be used within functions to store multiple values.

Example: Array in Function

```
sh
#!/bin/bash
print_fruits() {
    local fruits=("apple" "banana" "cherry")
    for fruit in "${fruits[@]}"; do
        echo "$fruit"
    done
}
print_fruits
```

This function print_fruits defines an array fruits and iterates over its elements.

Passing Arrays to Functions

Arrays can also be passed to functions by reference using positional parameters.

```
sh
#!/bin/bash
print_array() {
    local array=("$@")
    for element in "${array[@]}"; do
        echo "$element"
    done
}
fruits=("apple" "banana" "cherry")
print_array "${fruits[@]}"
```

In this example, the array fruits is passed to the print_array function.

Function Recursion

Recursion occurs when a function calls itself. This is useful for tasks that can be broken down into similar subtasks, like calculating factorials or navigating directory trees.

Example: Factorial Calculation

```
sh
#!/bin/bash
factorial() {
    if [ $1 -le 1 ]; then
        echo 1
    else
        local prev=$(factorial $(( $1 - 1 )))
        echo $(( $1 * prev ))
    fi
}
result=$(factorial 5)
echo "Factorial of 5 is $result"
```

This factorial function calculates the factorial of a given number using recursion.

Creating a Library

Creating Reusable Function Libraries

To make functions reusable across multiple scripts, you can place them in a separate file (library) and source it in your scripts.

Example: Function Library

Create a file named my_functions.sh:

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

```
    echo $(( $1 + $2 ))
}
```

Creating Text Menus

Text menus provide a simple, interactive interface for users to select options. They are commonly used in command-line scripts to improve usability and guide the user through various choices without requiring command-line knowledge.

Creating Basic Text Menus:

Using the select command in Bash is one of the easiest ways to create a text menu. The select command prompts the user with a list of options and waits for input.

Example:

```
sh
#!/bin/bash
PS3='Please enter your choice: '
options=("Option 1" "Option 2" "Option 3" "Quit")
select opt in "${options[@]}"
do
    case $opt in
        "Option 1")
            echo "You chose option 1"
            ;;
        "Option 2")
            echo "You chose option 2"
            ;;
        "Option 3")
            echo "You chose option 3"
            ;;
        "Quit")
            break
            ;;
          *) echo "Invalid option $REPLY";;
        esac
      done
```

Advanced Text Menus with Dialog

dialog is a utility that creates graphical user interface-like dialog boxes from shell scripts. It enhances the interactivity and visual appeal of text menus.

Installation:

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

Creating a Menu with Dialog:

```
sh
#!/bin/bash
HEIGHT=15
WIDTH=40
CHOICE_HEIGHT=4
BACKTITLE="Dialog Menu Example"
TITLE="My Menu"
MENU="Choose one of the following options:"
OPTIONS=("Option 1" "Description 1"
     "Option 2" "Description 2"
     "Option 3" "Description 3"
     "Quit" "Exit the script")
CHOICE=$(dialog --clear \
        --backtitle "$BACKTITLE" \
        --title "$TITLE" \
        --menu "$MENU" \
```

```
        $HEIGHT $WIDTH $CHOICE_HEIGHT \
        "${OPTIONS[@]}" \
        2>&1 >/dev/tty)
clear
case $CHOICE in
  "Option 1")
    echo "You chose option 1"
    ;;
  "Option 2")
    echo "You chose option 2"
    ;;
  "Option 3")
    echo "You chose option 3"
    ;;
  "Quit")
    exit 0
    ;;
esac
```

Building Text Window Widgets

Text window widgets provide a more interactive way for users to input data and receive feedback. They are useful for creating forms, dialogs, and other interactive text-based interfaces.

Using Dialog for Text Window Widgets

The dialog utility can create various types of text window widgets, such as input boxes, message boxes, and file selection dialogs.

Creating an Input Box:

```
sh

#!/bin/bash

dialog --title "Inputbox" --backtitle "Dialog Example" --inputbox "Enter
your name:" 8 40 2>input.txt

name=$(<input.txt)

echo "Hello, $name"
```

Creating a Message Box:

```
sh

#!/bin/bash

dialog --title "Message" --backtitle "Dialog Example" --msgbox "This is a
message box" 8 40
```

Creating a File Selection Box:

```
sh

#!/bin/bash

dialog --title "File Selection" --backtitle "Dialog Example" --fselect
$HOME/ 14 48 2>file.txt

file=$(<file.txt)

echo "You selected: $file"
```

Adding X Window Graphics

The X Window System provides a graphical windowing system for bitmap displays. It is the standard graphical interface for Unix-like operating systems, allowing for complex graphical applications.

Using Zenity for GUI Components

Zenity is a tool that allows you to display GTK+ dialog boxes from shell scripts. It provides a wide range of dialogs, including file selection, calendar, list, progress, and error dialogs.

Installation:

```
sh

sudo apt-get install zenity
```

Creating a Zenity Dialog:

```
sh

#!/bin/bash

zenity --info --text="Hello, World!"
```

Using Zenity for File Selection:

```
sh

#!/bin/bash

FILE=$(zenity --file-selection --title="Select a file")

echo "You selected: $FILE"
```

Using Zenity for Progress Dialogs:

```
sh

#!/bin/bash

(

for i in {1..100}; do

  echo $i

  sleep 0.1

done

) | zenity --progress --title="Progress Dialog" --text="Processing..." --
percentage=0 --auto-close
```

Combining Text Menus, Window Widgets, and X Window Graphics

Combining these elements allows you to create powerful and interactive scripts that can handle complex tasks while providing a user-friendly interface.

Example: Comprehensive Script Combining All Elements:

```
sh
#!/bin/bash
show_menu() {
  HEIGHT=15
  WIDTH=40
  CHOICE_HEIGHT=4
  BACKTITLE="Comprehensive Script"
  TITLE="Main Menu"
  MENU="Choose one of the following options:"
```

```
   OPTIONS=("Show Info" "Display Info Dialog"
            "Select File" "File Selection Dialog"
            "Progress" "Show Progress Dialog"
            "Quit" "Exit the script")
   CHOICE=$(dialog --clear \
                --backtitle "$BACKTITLE" \
                --title "$TITLE" \
                --menu "$MENU" \
                $HEIGHT $WIDTH $CHOICE_HEIGHT \
                "${OPTIONS[@]}" \
                2>&1 >/dev/tty)
   echo $CHOICE
}
handle_choice() {
   case $1 in
      "Show Info")
         zenity --info --text="This is an information dialog."
         ;;
      "Select File")
         FILE=$(zenity --file-selection --title="Select a file")
         zenity --info --text="You selected: $FILE"
         ;;
      "Progress")
         (
         for i in {1..100}; do
            echo $i
            sleep 0.1
         done
         ) | zenity --progress --title="Progress Dialog" --text="Processing..." --
percentage=0 --auto-close
         ;;
      "Quit")
         exit 0
         ;;
      *)
         zenity --error --text="Invalid choice."
         ;;
   esac
}
while true; do
   choice=$(show_menu)
   handle_choice "$choice"
done
```

Learning About the sed Editor

sed, short for Stream EDitor, is a non-interactive command-line utility used for parsing and transforming text. Unlike text editors like vim or nano, sed edits files in a non-interactive manner, making it ideal for automated text processing tasks.

Basic Concepts of sed

Commands: sed operates using commands that specify what actions to perform on the input text.

- **Patterns:** These are used to match specific parts of the text.
- **Addresses:** Define the scope or range within which the commands should apply.

Commonly Used Commands

Substitute (s): Used to replace occurrences of a pattern with a specified replacement.

```
sh
sed 's/old/new/' file.txt
```

This replaces the first occurrence of "old" with "new" on each line of file.txt.

Delete (d): Removes lines matching a pattern.

```
sh
sed '/pattern/d' file.txt
```

This deletes all lines containing "pattern" from file.txt.

Print (p): Outputs lines matching a pattern.

```
sh
sed -n '/pattern/p' file.txt
```

The -n option suppresses automatic printing, so only lines containing "pattern" are printed.

Examples of sed Usage

Replacing Text Globally:

```
sh
sed 's/old/new/g' file.txt
```

The g flag makes the substitution global, replacing all occurrences of "old" with "new" on each line.

In-Place Editing:

```
sh
sed -i 's/old/new/' file.txt
```

The -i option edits the file in place, directly modifying file.txt.

Getting Introduced to the gawk Editor

gawk is the GNU implementation of awk, a powerful programming language designed for pattern scanning and processing. awk is used to manipulate text based on patterns and is particularly useful for working with structured data, such as CSV files.

Basic Structure of gawk

An awk program consists of a sequence of pattern-action statements:

```sh
pattern { action }
```

Pattern: Specifies when the action is executed.
Action: A set of commands executed when the pattern matches.

Commonly Used Patterns and Actions

Pattern Matching: Perform actions based on matching patterns.

```sh
awk '/pattern/ { action }' file.txt
```

This performs action on lines matching "pattern".

Field and Record Processing:

$N: Refers to the Nth field of the current record (line).

```sh
awk '{ print $1, $2 }' file.txt
```

This prints the first and second fields of each line in file.txt.

NR: Built-in variable representing the record number.

```sh
awk '{ print NR, $0 }' file.txt
```

This prints the line number and the line content.

Examples of gawk Usage
Summing a Column of Numbers:

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

This script calculates the sum of the numbers in the first column of file.txt.

Finding Maximum Value:

```sh
awk 'BEGIN { max = 0 } { if ($1 > max) max = $1 } END { print max }'
file.txt
```

This finds the maximum value in the first column of file.txt.

Exploring sed Editor Basics

Inserting and Appending Lines

Insert (i): Adds a line before the matched pattern.

```sh
sed '/pattern/i\New line' file.txt
```

Append (a): Adds a line after the matched pattern.

```sh
sed '/pattern/a\New line' file.txt
```

Transforming Text

Uppercase Transformation:

```sh
sed 's/.*/\U&/' file.txt
```

This transforms each line to uppercase.

Lowercase Transformation:

```sh
sed 's/.*/\L&/' file.txt
```

Combining Multiple Commands

You can combine multiple sed commands using the -e option:

```sh
sed -e 's/old/new/' -e '/pattern/d' file.txt
```

This combines substitution and deletion in a single command.

UNIT 4

Regular Expressions:

Regular expressions, or regex, are sequences of characters that define a search pattern. These patterns are used for string matching within texts. Regex is incredibly useful for data validation, parsing, and transformation.

- **Efficiency:** Allows for efficient search and manipulation of strings.
- **Versatility:** Can be used across various programming languages and tools.
- **Power:** Provides powerful and flexible pattern matching capabilities.

Basic Syntax:

Literals: Matches the exact character or string.

```
sh
grep "hello" file.txt  # Matches "hello"
```

Metacharacters:

Special characters with specific meanings.

.	: Matches any single character except newline.
^	: Matches the start of a line.
$	: Matches the end of a line.
*	: Matches zero or more of the preceding character.
+	: Matches one or more of the preceding character.
?	: Matches zero or one of the preceding character.
[]	: Matches any one of the characters inside the brackets.
|	: Logical OR.
()	: Groups patterns together.

Examples of Basic Patterns:

Single Character Match:

```
sh
grep ".at" file.txt  # Matches "cat", "bat", "hat", etc.
```

Line Start and End:

```
sh
grep "^start" file.txt  # Matches lines that start with "start"
grep "end$" file.txt    # Matches lines that end with "end"
```

Character Classes:

```
sh
grep "[abc]" file.txt  # Matches any line containing "a", "b", or "c"
grep "[0-9]" file.txt  # Matches any digit
```

Extending Our Patterns:

Quantifiers:

Quantifiers specify how many times a character or group should be matched.

{n}: Exactly n times.

```
sh
grep "a{3}" file.txt  # Matches "aaa"
```

{n,}: At least n times.

```
sh
grep "a{3,}" file.txt  # Matches "aaa", "aaaa", etc.
```

{n,m}: Between n and m times.

```
sh
grep "a{2,4}" file.txt  # Matches "aa", "aaa", "aaaa"
```

Anchors and Boundaries

Word Boundaries: \b matches a word boundary.

```
sh
grep "\bword\b" file.txt  # Matches "word" as a whole word
```

Negation: ^ inside square brackets negates the pattern.

```
sh

grep "[^0-9]" file.txt  # Matches any character that is not a digit
```

Creating Expressions:

Combining Patterns: Combining simple patterns to create complex expressions:

Alternation: Using | to specify multiple patterns.

```
sh

grep "cat|dog" file.txt  # Matches lines containing "cat" or "dog"
```

Groups and Backreferences: Using () to group patterns and refer back to them.

```
sh

grep "\(foo\).*\1" file.txt  # Matches "foo" followed by any characters
and "foo" again
```

Advanced Techniques

Look ahead and Look behind: Asserts that a match is followed or preceded by another pattern without including it in the match.

Positive Lookahead: (?=...)

```
sh

grep -P 'foo(?=bar)' file.txt  # Matches "foo" only if followed by "bar"
```

Negative Lookahead: (?!...)

```
sh

grep -P 'foo(?!bar)' file.txt  # Matches "foo" only if not followed by "bar"
```

Positive Lookbehind: (?<=...)

```
sh
```

```
grep -P '(?<=foo)bar' file.txt  # Matches "bar" only if preceded by "foo"
```

Negative Lookbehind: (?<!...)

```
sh

grep -P '(?<!foo)bar' file.txt  # Matches "bar" only if not preceded by "foo"
```

Practical Applications

1. Text Processing

Finding and Replacing Text:

```
sh

sed -i 's/oldtext/newtext/g' file.txt  # Replaces "oldtext" with "newtext"
```

2. Data Validation

Email Validation:

```
sh

echo "test@example.com" | grep -E "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-
Z]{2,}$"
```

3. Log File Analysis

Extracting IP Addresses:

```
sh

grep -oE '([0-9]{1,3}\.){3}[0-9]{1,3}' logfile.txt
```

Advanced sed

Using Multiline Commands

sed typically operates on a single line of text at a time. However, there are situations where processing multiple lines together is necessary, such as when patterns span across lines.

Example: Joining Lines

To join every pair of lines in a file, you can use the N command which appends the next line to the pattern space:

```
sh
sed 'N;s/\n/ /' file.txt
```

This command appends the next line to the pattern space and then replaces the newline character with a space, effectively joining the lines.

Example: Paragraph Processing

To match patterns that span multiple lines, you might need the H, G, and x commands to handle the hold space.

```
sh
sed '/./{H;$!d;} ; x ; s/\n/ /g' file.txt
```

This script collects paragraphs (blocks of text separated by blank lines) and replaces newlines within each paragraph with spaces.

Understanding the Hold Space

The hold space is an additional buffer that sed provides, allowing you to store and retrieve text for later use. It’s useful for tasks that require temporary storage and manipulation.

Example: Swapping Lines

Using the hold space to swap adjacent lines:

```
sh
sed 'N;h;n;G' file.txt
```

This command sequence performs the following:

N: Appends the next line to the pattern space.

H: Copies the pattern space to the hold space.

n: Moves to the next line.

G: Appends the hold space to the pattern space.

Example: Conditional Replacement

Using hold space for more complex replacements:

```
sh
sed -n 'H; $ { x; s/foo/bar/g; p; }' file.txt
```

This example accumulates all lines in the hold space and performs a global substitution (s/foo/bar/g) on the entire file content at the end.

Negating a Command

Negating a command in sed allows you to apply actions to lines that do not match a specific pattern. This is useful for excluding certain lines from processing.

Example: Excluding Lines

To print all lines except those that contain the word "error":

```
sh
sed '/error/!p' file.txt
```

The ! operator negates the match, so the p command only applies to lines that do not match "error".

Example: Selective Replacement

Replacing all occurrences of "foo" with "bar" except on lines that start with "#":

```
sh
sed '/^#/!s/foo/bar/g' file.txt
```

This ensures that the substitution only occurs on non-comment lines (assuming "#" denotes a comment).

Changing the Flow

sed provides commands for altering the flow of its script, allowing you to control which parts of the script are executed based on various conditions.

Example: Branching

Using the b command for unconditional branching and t for conditional branching.

```
sh
sed '/start/b jump; /skip/d; :jump; s/foo/bar/g' file.txt
```

This script:

Branches to the label jump if the line contains "start".

Deletes lines containing "skip".

Replaces "foo" with "bar" starting from the label jump.

Example: Conditional Branching

Using the t command to branch if a substitution was made:

```
sh
sed -e 's/foo/bar/; t label; s/baz/qux/; :label' file.txt
```

Here, the script performs a substitution s/foo/bar/ and branches to label if the substitution is successful, skipping s/baz/qux/.

Replacing via a Pattern

Pattern-based replacements in sed allow for highly targeted text transformations based on complex patterns.

Example: Simple Pattern Replacement

Replace all instances of "apple" with "orange":

```
sh
sed 's/apple/orange/g' file.txt
```

Example: Contextual Replacement

Replace "foo" with "bar" only if it appears between "start" and "end":

```
sh
sed '/start/,/end/s/foo/bar/g' file.txt
```

This confines the replacement to the range between lines matching "start" and "end".

Using sed in Scripts

Integrating sed into shell scripts enables automated text processing tasks, enhancing the script's capabilities.

Example: Batch Processing Files

A script to replace a pattern in multiple files:

```
sh
#!/bin/bash
for file in *.txt; do
  sed -i 's/foo/bar/g' "$file"
done
```

This script iterates over all .txt files in the directory and performs an in-place substitution.

Example: Dynamic Pattern Replacement

A script to replace a pattern provided as an argument:

```
sh
#!/bin/bash
pattern=$1
replacement=$2
file=$3
sed -i "s/$pattern/$replacement/g" "$file"
```

Usage:

```
sh
./script.sh foo bar file.txt
```

This replaces all instances of "foo" with "bar" in file.txt.

Creation of sed Utilities

Creating reusable sed utilities can streamline repetitive tasks and enhance productivity.

Example: Line Numberer

A sed script to add line numbers to a file:

```
sh
#!/bin/sed -nf
```

=	# Print the current line number
N	# Append the next line to the pattern space
s/\n/: /	# Replace the newline character with a colon and space
P	# Print the current pattern space
D	# Delete the pattern space and start a new cycle

Usage:

```
sh
./line_numberer.sed file.txt
```

Example: Extracting Sections

Extracting sections of a file based on patterns:

```
sh
#!/bin/sed -nf
/start/,/end/p
```

Advanced gawk

Reexamining gawk

gawk

gawk is an extended version of the awk programming language. It stands for GNU awk and includes additional features that make it more powerful and versatile for text manipulation, pattern scanning, and report generation. awk processes data line by line, applying patterns and actions specified by the user.

Basic Structure of an awk Script

An awk program consists of a sequence of pattern-action statements:

```
awk
pattern { action }
```

- **Pattern:** Specifies when the action is executed.
- **Action:** A set of commands executed when the pattern matches.

Using Variables in gawk

Variables in gawk can store data, such as numbers and strings, for use in calculations, comparisons, and other operations. They can be predefined (built-in) or user-defined.

Built-in Variables

Some of the commonly used built-in variables include:

NR:	Current record number.
NF:	Number of fields in the current record.
FS:	Field separator (default is space).
RS:	Record separator (default is newline).
OFS:	Output field separator.
ORS:	Output record separator.

User-defined Variables

You can define your own variables and assign values to them:

```
awk
BEGIN {
   count = 0
   total = 0
}
{
   count++
   total += $1
}
END {
   print "Count:", count
   print "Total:", total }
```

This script counts the number of lines and sums the values in the first column.

Using Structured Commands

Control Structures

awk supports various control structures, including conditional statements and loops, similar to other programming languages.

Conditional Statements

if-else Statement:

```
awk
{
    if ($1 > 50)
        print $1, "is greater than 50"
    else
        print $1, "is less than or equal to 50"
}
```

if-else if-else Statement:

```
awk
{
    if ($1 > 75)
        print $1, "is greater than 75"
    else if ($1 > 50)
        print $1, "is greater than 50 but less than or equal to 75"
    else
        print $1, "is less than or equal to 50"
}
```

Loops

while Loop:

```
awk
BEGIN {
    i = 1
    while (i <= 10) {
        print i
        i++
    }
}
```

for Loop:

```
awk
BEGIN {
    for (i = 1; i <= 10; i++)
        print i
}
```

Arrays

Arrays in awk are associative, meaning they can be indexed by strings as well as numbers.

```
awk
BEGIN {
    fruits["apple"] = 1
    fruits["banana"] = 2
    fruits["cherry"] = 3
    for (fruit in fruits)
        print fruit, fruits[fruit]
}
```

Formatting the Printing Work

Print and printf

gawk provides print and printf for output formatting.

print Statement:

```
awk
{
   print "Name:", $1, "Age:", $2
}
```

This statement prints fields with a space separator.

printf Statement:

```
awk
{
   printf "Name: %s, Age: %d\n", $1, $2
}
```

printf allows for formatted output similar to C's printf, offering more control over the output format.

Field Separators and Formatting

Customizing field separators and formats enhances the readability of output:

Field Separator:

```
awk
BEGIN { FS = "," }
{
   print $1, $2, $3
}
```

Output Formatting:

```
awk
BEGIN { OFS = " | " }
{
   print $1, $2, $3
}
```

Working with Functions

Built-in Functions

gawk provides numerous built-in functions for mathematical operations, string manipulation, and more.

Mathematical Functions:

sin(), cos(), sqrt(), log(), etc.

```
awk
{
   print "Square root of", $1, "is", sqrt($1)
}
```

String Functions:

length(), substr(), index(), toupper(), tolower(), etc.

```
awk
{
   print "Length of", $1, "is", length($1)
   print "Substring of", $1, "from index 2 to 4 is", substr($1, 2, 3)
}
```

User-defined Functions

You can define your own functions in awk to encapsulate reusable logic.

```
awk
function factorial(n) {
   if (n <= 1)
```

```
        return 1
    else
        return n * factorial(n - 1)
}
BEGIN {
    print "Factorial of 5 is", factorial(5)
}
```

UNIT 5

Understanding the Dash Shell

The Dash (Debian Almquist Shell) is a streamlined, lightweight POSIX-compliant shell. It's predominantly utilized as the default /bin/sh on Debian-based systems due to its efficiency and small footprint. But why would one opt for Dash over the ubiquitous Bash shell? Let's break it down.

Characteristics of Dash

- **Performance-Oriented:** Dash is significantly faster than Bash for running shell scripts, making it a prime candidate for environments where performance is critical.
- **Minimalistic:** Dash is a no-frills shell that adheres closely to POSIX standards, foregoing many of Bash's extensions to remain lightweight.
- **Portability:** Scripts written for Dash are typically more portable across different Unix-like systems since they stick to POSIX features.

Here's a simple example of a script written in Dash:

```
#!/bin/dash
echo "This script is running in Dash"
```

Notice how similar it looks to Bash. That's the beauty of POSIX compliance; it ensures a level of consistency across different environments.

Differences from Bash

Dash lacks some of Bash's advanced features like arrays and certain built-in commands. This minimalism is both a strength and a limitation. While it ensures scripts are lean and mean, it may require rethinking certain approaches if you're used to Bash-specific functionalities.

The Zsh Shell

On the other end of the spectrum is Zsh (Z Shell), which is renowned for its feature richness and customizability. It's the shell of choice for many power users, providing extensive capabilities that can be tailored to fit a myriad of needs.

Characteristics of Zsh:

- **Customizability:** Zsh is immensely customizable. With the oh-my-zsh framework, users can apply themes and plugins that drastically enhance the command-line experience.
- **Advanced Scripting:** Zsh supports more advanced programming constructs than many other shells, including associative arrays and floating-point arithmetic.
- **Interactive Features:** Zsh offers superior auto-completion, spell correction, and command-line editing, making it highly interactive.

Here's an example of a script utilizing Zsh's advanced features:

Zsh

```
#!/bin/zsh
# Using an associative array in Zsh
typeset -A fruits
fruits[apple]=100
fruits[banana]=150
fruits[orange]=200
echo "Price of apple is: ${fruits[apple]}"
```

This script demonstrates how Zsh's advanced features, like associative arrays, can simplify complex tasks.

Writing Scripts in Alternative Shells:

Whether you choose Dash or Zsh, writing scripts in these shells can offer unique advantages.

Scripting in Dash:

When scripting in Dash, adhering to POSIX standards is crucial. This ensures your scripts are portable and can run on any POSIX-compliant system. Avoid Bash-specific constructs and extensions.

Example Dash Script:

```
sh
#!/bin/dash
if [ "$#" -lt 1 ]; then
  echo "Usage: $0 argument"
  exit 1
fi
echo "Argument provided: $1"
```

This script checks if an argument is provided and prints it, demonstrating basic error handling and argument parsing.

Scripting in Zsh :

When scripting in Zsh, you can leverage its advanced features for more sophisticated scripts. Utilize associative arrays, floating-point arithmetic, and enhanced completion to create powerful scripts.

Example Zsh Script:

```
zsh
#!/bin/zsh
```

```
a=10.5
b=2.3
result=$(echo "$a + $b" | bc)
echo "The result of $a + $b is $result"
```

This script shows how Zsh handles floating-point arithmetic, showcasing its advanced capabilities compared to Dash.

Choosing the Right Shell:

Choosing between Dash and Zsh depends on your specific needs and environment.

Performance and Minimalism: Dash

- **When to Use:** Opt for Dash when performance is paramount, and you need a lightweight shell that adheres to strict POSIX standards. Ideal for simple, portable scripts that need to run efficiently.
- **Example Use Case:** Automating system tasks on a server where resources are limited, and speed is crucial.

Customizability and Advanced Features: Zsh

- **When to Use:** Choose Zsh when you need a powerful, feature-rich shell with extensive customization options. Perfect for complex scripting tasks and interactive use.
- **Example Use Case:** Enhancing your development environment with plugins and themes that streamline your workflow.

Enhancing Your Scripting Skills

Exploring alternative shells not only broadens your scripting skills but also provides a deeper understanding of the Unix/Linux environment. By experimenting with Dash and Zsh, you can:

- **Improve Efficiency:** Writing optimized scripts that execute faster in Dash.
- **Enhance Productivity:** Utilizing Zsh's advanced features and customization to streamline your daily tasks.

Exercise 1: Writing a Dash Script

Objective: Write a script in Dash to list all files in a directory and output their sizes.

Steps:

1. Use POSIX-compliant commands.
2. Ensure the script is portable across different Unix-like systems.
3. Test the script in a Debian-based environment.

Example Solution:

```
sh
#!/bin/dash
# Check if directory is provided
if [ "$#" -ne 1 ]; then
  echo "Usage: $0 directory"
  exit 1
fi
for file in "$1"/*; do
  if [ -f "$file" ]; then
    size=$(wc -c < "$file")
    echo "File: $file, Size: $size bytes"
  fi
done
```

Exercise 2: Writing a Zsh Script

Objective: Write a script in Zsh to manage a to-do list.

Steps:

1. Use associative arrays to store tasks and their statuses.
2. Implement functions to add, remove, and list tasks.
3. Utilize Zsh's advanced scripting features for efficiency.

Example Solution:

```
zsh
#!/bin/zsh
typeset -A tasks
```

Scripting Utilities

Scripting utilities are an essential tool in any programmer or system administrator's toolkit. They automate repetitive tasks, streamline workflows, and enhance productivity by reducing manual intervention. In this comprehensive guide, we will explore the creation and usage of simple scripting utilities, covering various scenarios such as automating backups, managing user accounts, and monitoring disk space. By the end of this guide, you'll have a solid foundation to build your own useful scripts to automate daily tasks.

Automating Backups

Backups are crucial for data integrity and disaster recovery. Automating backups ensures that critical data is consistently saved without requiring manual intervention. Here's how you can automate backups using shell scripting.

1. Basic Backup Script:

A simple script to back up a directory can be created using the following steps:

Identify the Source and Destination: Define the directory to be backed up and the backup destination.

Use rsync for Efficient Copying: The rsync command is highly efficient for incremental backups, copying only changed files.

```
bash
#!/bin/bash
SOURCE_DIR="/path/to/source"
BACKUP_DIR="/path/to/backup"
TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")
BACKUP_PATH="$BACKUP_DIR/backup_$TIMESTAMP"
echo "Starting backup of $SOURCE_DIR to $BACKUP_PATH"
rsync -av --delete "$SOURCE_DIR/" "$BACKUP_PATH/"
echo "Backup completed at $BACKUP_PATH"
```

2. Scheduling Backups with Cron :

To ensure backups run at regular intervals, you can schedule the backup script using cron jobs.

```
bash
# Open the crontab file
crontab -e
# Add a cron job to run the backup script daily at 2 AM
0 2 * * * /path/to/backup_script.sh
```

Managing User Accounts

Managing user accounts can be tedious, especially on systems with a large number of users. Scripting utilities can simplify tasks like creating, modifying, and deleting user accounts.

1. Creating User Accounts

The following script automates the creation of new user accounts, including setting passwords and creating home directories.

```
bash
#!/bin/bash
USERNAME=$1
PASSWORD=$2
if [ -z "$USERNAME" ] || [ -z "$PASSWORD" ]; then
    echo "Usage: $0 username password"
    exit 1
fi
echo "Creating user: $USERNAME"
useradd -m -s /bin/bash "$USERNAME"
echo "$USERNAME:$PASSWORD" | chpasswd
echo "User $USERNAME created successfully"
```

2. Deleting User Accounts

A script to delete user accounts can be similarly created to ensure accounts are removed cleanly and home directories are purged.

```
bash
#!/bin/bash
USERNAME=$1
if [ -z "$USERNAME" ]; then
    echo "Usage: $0 username"
    exit 1
fi
echo "Deleting user: $USERNAME"
userdel -r "$USERNAME"
echo "User $USERNAME deleted successfully"
```

Watching Disk Space

Monitoring disk space is vital to prevent systems from running out of storage, which can lead to crashes and data loss. A scripting utility can be created to monitor disk usage and alert administrators when space is running low.

1. Disk Space Monitoring Script

This script checks the disk usage of specified partitions and sends an email alert if usage exceeds a predefined threshold.

```
bash
#!/bin/bash
THRESHOLD=80
PARTITIONS=("/" "/home")
for PARTITION in "${PARTITIONS[@]}"; do
    USAGE=$(df -h "$PARTITION" | grep -v Filesystem | awk '{print $5}' | sed 's/%//g')
    if [ "$USAGE" -gt "$THRESHOLD" ]; then
        echo "Disk space critical on $PARTITION ($USAGE%)" | mail -s "Disk Space
Alert: $PARTITION" admin@example.com
    fi
done
```

2. Scheduling Disk Space Monitoring

As with backups, you can schedule the disk space monitoring script to run at regular intervals using cron.

```
bash
# Open the crontab file
crontab -e
# Add a cron job to run the disk space monitoring script every hour
0 * * * * /path/to/disk_space_monitor.sh
```

Producing Scripts for Database, Web, and E-Mail

Writing Database Shell Scripts:

Database management often requires repetitive tasks such as backups, data imports, and maintenance. Scripting these tasks can save significant time and reduce errors.

1. Database Backup Script:

A script to back up a MySQL database can be created as follows:

```
bash
#!/bin/bash
DB_NAME="mydatabase"
DB_USER="myuser"
DB_PASS="mypassword"
BACKUP_DIR="/path/to/backup"
TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")
BACKUP_PATH="$BACKUP_DIR/$DB_NAME_$TIMESTAMP.sql"
echo "Starting backup of $DB_NAME"
mysqldump -u $DB_USER -p$DB_PASS $DB_NAME > $BACKUP_PATH
echo "Backup completed at $BACKUP_PATH"
```

2. Importing Data into Database:

A script to import data from a file into a MySQL database:

```
bash
#!/bin/bash
DB_NAME="mydatabase"
DB_USER="myuser"
DB_PASS="mypassword"
DATA_FILE="/path/to/datafile.sql"
echo "Importing data into $DB_NAME"
mysql -u $DB_USER -p$DB_PASS $DB_NAME < $DATA_FILE
echo "Data import completed"
```

Using the Internet from Your Scripts:

With the internet's power, you can integrate web services, download data, and even interact with APIs directly from your scripts.

1. Downloading Files

A script to download a file from the internet:

```
bash
#!/bin/bash
URL="https://example.com/file.txt"
DESTINATION="/path/to/downloaded_file.txt"
echo "Downloading $URL"
curl -o $DESTINATION $URL
echo "File downloaded to $DESTINATION"
```

2. Fetching API Data:

A script to fetch data from an API and process it:

```
bash
#!/bin/bash
API_URL="https://api.example.com/data"
API_KEY="your_api_key"
echo "Fetching data from $API_URL"
curl -H "Authorization
```

Email Reports:

Automation of email reports is crucial for system administrators, developers, and anyone who needs to send regular updates, logs, or notifications. This practice ensures:

- **Consistency:** Regular tasks are executed without fail.
- **Timeliness:** Reports are sent at scheduled times without human intervention.
- **Efficiency:** Reduces manual work, allowing time for more complex tasks.

How Does It Work?

Email automation involves scripting with tools available in the Linux environment. Key components include:

- **Mail Transfer Agent (MTA):** Software like sendmail, postfix, or mailutils that sends emails from the server.
- **Scheduling Tool:** cron for scheduling the script to run at specified intervals.

- **Script:** The script itself, written in shell scripting language, handles generating the report and sending the email.

Setting Up the Environment

First, ensure you have mailutils installed for sending emails.

```
sh
sudo apt-get install mailutils
```

Writing a Basic Script

Creating a Report Script: This script generates a system report and saves it to a file.

```
sh
#!/bin/bash
# Generate a system report
echo "System Report" > report.txt
echo "Date: $(date)" >> report.txt
echo "Uptime: $(uptime)" >> report.txt
echo "Disk Usage:" >> report.txt
df -h >> report.txt
```

Emailing the Report: This script sends the generated report via email.

```
sh
#!/bin/bash
TO_EMAIL="recipient@example.com"
SUBJECT="Daily System Report"
# Generate the report
echo "System Report" > report.txt
echo "Date: $(date)" >> report.txt
echo "Uptime: $(uptime)" >> report.txt
echo "Disk Usage:" >> report.txt
df -h >> report.txt
# Send the report
```

```
mail -s "$SUBJECT" "$TO_EMAIL" < report.txt
```

Scheduling the Script with cron

To ensure the script runs automatically at specified times, use cron.

```
sh
# Open the crontab file
crontab -e
# Add the following line to run the script daily at 8 AM
0 8 * * * /path/to/your_script.sh
```

Enhancing the Script

Sending Emails with Attachments: If you need to send the report as an attachment, use mutt.

```
sh
sudo apt-get install mutt
sh
#!/bin/bash
TO_EMAIL="recipient@example.com"
SUBJECT="System Report with Attachment"
BODY="Please find the attached system report."
ATTACHMENT="report.txt"
# Generate the report
echo "System Report" > report.txt
echo "Date: $(date)" >> report.txt
echo "Uptime: $(uptime)" >> report.txt
echo "Disk Usage:" >> report.txt
df -h >> report.txt
# Send the email with attachment
echo "$BODY" | mutt -s "$SUBJECT" -a "$ATTACHMENT" -- "$TO_EMAIL"
```

Technical Requirements

1. Python Installation

To use Python for scripting, ensure it's installed on your system. Most Unix-like systems come with Python pre-installed. For Windows, you can download it from the official Python website.

```
sh
# Check if Python is installed
python –version
```

2. Text Editor:

You'll need a text editor to write your scripts. Popular choices include:

- **VS Code:** Versatile and packed with features.
- **PyCharm:** Specifically designed for Python.
- **Vim/Emacs:** For the command-line purists.

Python Language

1. Why Python?:

Python is known for its readability and simplicity. Its syntax is clear, making it easy to learn and use. It also has a large standard library that supports many common programming tasks such as connecting to web servers, reading and modifying files, and working with data.

2. Python vs Bash:

- **Readability:** Python's syntax is more readable and straightforward than Bash.
- **Portability:** Python scripts can run on any operating system with Python installed, whereas Bash scripts are primarily Unix/Linux-focused.
- **Functionality:** Python has extensive libraries and frameworks that make it powerful for various tasks beyond shell scripting.

```
python
print("Hello, World!")
```

To execute this script, save it as hello.py and run:

```
sh
python hello.py
```

Pythonic Arguments

1. Understanding Pythonic Arguments

In Python, arguments are passed to scripts in a way that's both easy to understand and use. This is facilitated by modules like argparse and sys.

2. Using sys.argv

sys.argv is a list in Python, which contains the command-line arguments passed to the script.

```
python
import sys
print("Number of arguments:", len(sys.argv))
print("Argument List:", str(sys.argv))
```

Supplying Arguments

1. Example with argparse

The argparse module makes it easy to write user-friendly command-line interfaces.

```
python
import argparse
# Create the parser
parser = argparse.ArgumentParser(description='Example with argparse')
# Add arguments
parser.add_argument('name', type=str, help='Name of the user')
parser.add_argument('age', type=int, help='Age of the user')
# Parse the arguments
args = parser.parse_args()
print(f"Hello {args.name}, you are {args.age} years old.")
```

Counting Arguments

1. Using sys.argv to Count Arguments

You can easily count the number of arguments passed to a Python script using sys.argv.

```
python
    import sys
    print(f"Total number of arguments passed: {len(sys.argv) - 1}")
```

Significant Whitespace

1. Indentation in Python:

Python uses indentation to define the scope of loops, functions, and classes. It's crucial to maintain consistent indentation to avoid syntax errors.

```
python
    def greet(name):
        if name:
            print(f"Hello, {name}!")
        else:
            print("Hello, World!")
```

Reading User Input:

1. Using input()

Python's input() function allows you to read user input.

```
python
    name = input("Enter your name: ")
    print(f"Hello, {name}!")
```

Using Python to Write to Files

1. Writing Data to a File

You can write data to files using Python's open() function.

```
python
    with open('example.txt', 'w') as file:
        file.write('Hello, World!\n')
        file.write('Writing to a file in Python.')
```

2. Reading Data from a File:

Reading data from a file is just as straightforward.

```
python
    with open('example.txt', 'r') as file:
        content = file.read()
        print(content)
```

String Manipulation:

1. Basic String Operations:

Python provides powerful string manipulation capabilities.

```
python
    string = "Hello, World!"
    print(string.upper())  # Convert to uppercase
    print(string.lower())  # Convert to lowercase
    print(string.split())  # Split string into a list
    print(string.replace("World", "Python"))  # Replace substring
```

2. Formatting Strings

Python supports various ways to format strings.

Using f-Strings (Python 3.6+):

```
python
    name = "Alice"
    age = 25
    print(f"My name is {name} and I am {age} years old.")
```

Using str.format():

```
python
    print("My name is {} and I am {} years old.".format(name, age))
```

Integrating Python with Shell Scripts

1. Running Shell Commands

You can run shell commands directly from Python using the subprocess module.

python

```
import subprocess
# Run a simple command
subprocess.run(["echo", "Hello, World!"])
# Capture the output of a command
result = subprocess.run(["ls", "-l"], capture_output=True, text=True)
print(result.stdout)
```

2. Combining Python and Bash

Sometimes, you might want to combine Python with existing Bash scripts.

Calling Python from Bash:

```
sh
#!/bin/bash
python3 -c 'print("Hello from Python!")'
```

Calling Bash from Python:

```
python
    import subprocess
    subprocess.run(["bash", "-c", "echo Hello from Bash!"])
```