Access the Command Line

Objectives

Log in to a Linux system and run simple commands with the shell.

Introduction to the Bash Shell

The command line is a text-based interface that allows users to communicate with a computer by entering instructions. In Linux, this interface is powered by a program known as the shell. While multiple shell options exist, Red Hat recommends sticking to the default shell for system administration.

The standard shell in Red Hat Enterprise Linux (RHEL) is the GNU Bourne-Again Shell (Bash), an advanced successor to the original UNIX Bourne Shell (sh).

When a shell session starts, a prompt appears, signaling readiness for user input:

- Regular users see a $ symbol ([user@host ~]$).
- Superusers (root) see a # symbol ([root@host ~]#), emphasizing the higher privileges and potential risks of administrative actions.

Why Use Bash?

Bash is a powerful tool for executing commands and scripting, making it possible to automate repetitive tasks. It streamlines complex operations that graphical tools may struggle to handle efficiently.

Comparison with Other Systems:

- **Windows:** Bash surpasses cmd.exe in capabilities and is more comparable to **PowerShell** due to its scripting flexibility.
- **macOS:** Before macOS 10.15 Catalina, Bash was the default shell. Apple later replaced it with zsh, which is also available in RHEL.

Understanding Shell Commands

A command entered in the shell typically consists of three components:

1. Command – The name of the program being executed.
2. Options – Modifications that alter the command's behavior (e.g., -L, --all).
3. Arguments – Specific targets the command should act upon.

Example Command:

```
usermod -L user01
```

- **usermod** = Command
- **-L** = Option (locks the user account)
- **user01** = Argument (target user)

Mastering Bash enables users to efficiently manage Linux systems, automate tasks, and improve workflow productivity.

Log in to a Local System

A terminal is a text-based interface that allows users to enter commands and receive output from a computer. To access a shell, users must log in through a terminal session.

A Linux machine may have a physical console, consisting of a directly connected keyboard and display. This console supports multiple virtual consoles, each capable of running independent terminal sessions. Users can switch between virtual consoles using Ctrl+Alt + Function keys (F1–F6). Most virtual consoles provide a text-based login prompt, where users can enter their credentials to access a shell session.

On systems with a graphical interface, the login screen appears on one of the virtual consoles. Once logged in, users must open a terminal emulator within the graphical environment to access the shell prompt.

Key Considerations for System Administrators:

- Many administrators prefer running servers without a graphical interface to conserve system resources, as servers typically do not function as desktop workspaces.
- In Red Hat Enterprise Linux 9, the graphical login screen runs on the first virtual console (tty1), while text-based logins are available on consoles tty2–tty6.
- Once a graphical session starts, it usually occupies the next available virtual console (commonly tty2) unless another session is already active.

Headless Servers and Remote Access:

A headless server operates without a dedicated monitor and keyboard, often used in data centers with multiple servers mounted in racks.

- These servers are typically accessed remotely through a serial console, which provides a login prompt via a serial port connected to a console server.
- If network connectivity is lost due to misconfiguration, the serial console remains a reliable way to regain access.

- In most cases, remote management tools like Virtual Network Computing (VNC) are used to run a graphical interface on the server when needed.

By understanding how to log into both local and remote systems, administrators can efficiently manage Linux environments, whether using direct console access or remote tools.

Log in to a Remote System

In Linux environments, users and administrators frequently need to establish remote shell access over a network. Many modern servers operate as virtual machines or cloud instances, often without a physical console or direct hardware access. In such cases, remote access is essential for management and administration.

Secure Shell (SSH): The Standard for Remote Access
The most widely used method for connecting to remote Linux systems is Secure Shell (SSH). Most Linux distributions, including Red Hat Enterprise Linux (RHEL), as well as macOS, provide the OpenSSH client, which includes the **ssh** command.

For example, a user on a local machine can connect to a remote system named **remotehost** using the **ssh** command:

```
[user@host ~]$ ssh remoteuser@remotehost
remoteuser@remotehost's password: password
[remoteuser@remotehost ~]$
```

The SSH protocol ensures that all communication, including login credentials, is encrypted to prevent unauthorized interception.

Passwordless Authentication with SSH Keys:
Some remote systems, particularly cloud instances, disable password authentication for security reasons. Instead, they rely on public key authentication, which eliminates the need to enter a password manually.

- The user generates an SSH key pair consisting of a private key (kept secret) and a public key (stored on the remote server).
- When the user attempts to log in, SSH uses the private key to authenticate against the stored public key.

To log in using an SSH private key file, the -i option is used:

```
[user@host ~]$ ssh -i mylab.pem remoteuser@remotehost
[remoteuser@remotehost ~]$
```

For security, the private key file must be readable only by its owner. This can be enforced with the following command:

```
chmod 600 mylab.pem
```

First-Time Login and SSH Host Key Verification:
When connecting to a remote system for the first time, SSH warns that the host authenticity cannot be verified:

```
[user@host ~]$ ssh -i mylab.pem remoteuser@remotehost
The authenticity of host 'remotehost (192.0.2.42)' can't be established.
ECDSA key fingerprint is 47:bf:82:cd:fa:68:06:ee:d8:83:03:1a:bb:29:14:a3.
Are you sure you want to continue connecting (yes/no)? yes
[remoteuser@remotehost ~]$
```

This message appears because SSH relies on host keys to verify that the remote system is genuine. If the key is unknown (i.e., not previously saved), SSH prompts the user to accept it.

- Entering "yes" saves the key and allows future connections without warning.
- Entering "no" cancels the connection.
- If the saved key changes unexpectedly, SSH will block the connection, as this could indicate a security risk (e.g., a man-in-the-middle attack).

By following these best practices for SSH authentication, users can securely manage remote Linux systems while minimizing security vulnerabilities.

Log Out from a Remote System

When you're done using the shell and want to end your session, there are multiple ways to log out. The simplest method is to type the **exit** command, which terminates the current shell session. Another quick way is pressing **Ctrl+D**, which signals the shell to close.

Below is an example of a user logging out from an SSH session:

```
[remoteuser@remotehost ~]$ exit
logout
Connection to remotehost closed.
[user@host ~]$
```

Additional Resources:
For further reading on terminal sessions, shell usage, and SSH security, you can explore the following references:

- man pages: intro(1), bash(1), pts(4), ssh(1), and ssh-keygen(1)
- Detailed guidance on OpenSSH and public key authentication is available in the Securing Networks section of the Red Hat Enterprise Linux 9 documentation: Red Hat Enterprise Linux 9 Securing Networks Guide

In an upcoming chapter, you'll also learn how to access and navigate man pages and other built-in Linux documentation effectively.

Access the Command Line with the Desktop

Objectives

Log in to the Linux system with the GNOME desktop environment to run commands from a shell prompt in a terminal program.

Introduction to the GNOME Desktop Environment

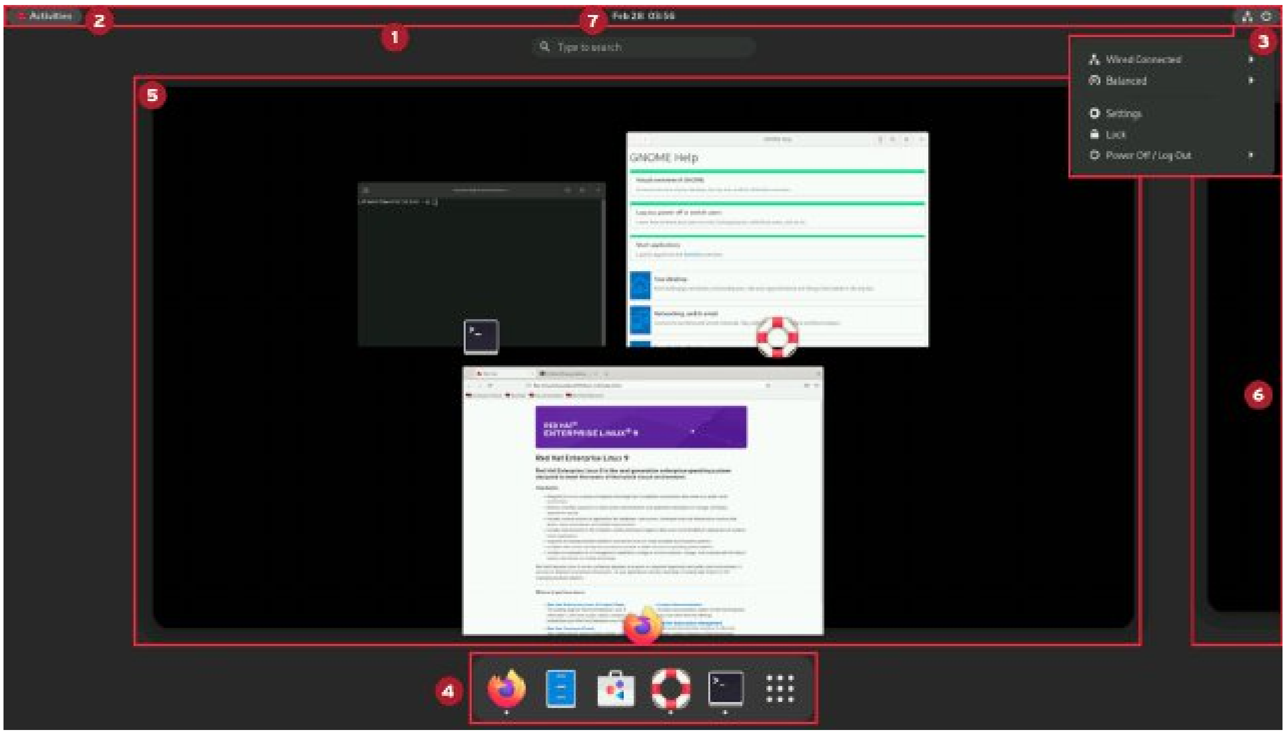
A desktop environment provides a graphical interface for interacting with a Linux system. In Red Hat Enterprise Linux 9, **GNOME 40** is the default desktop environment. It delivers a seamless user experience and serves as a unified platform for application development, built on either **Wayland** (the default) or the older **X Window System**.

At the heart of GNOME is GNOME Shell, which powers the main interface. It offers extensive customization options. By default, RHEL 9 uses the Standard theme, but you can switch to the Classic theme for a look similar to earlier versions of GNOME. You can change themes at login by selecting the gear icon after choosing your account but before entering your password.

New users are greeted with an optional "Take a Tour" program on their first login, which introduces key features of GNOME in RHEL 9.

To access GNOME Help, click the Activities button on the top-left, then select the Help icon from the application menu at the bottom.

Parts of the GNOME Shell



1) **Top Bar:**
   The top bar runs across the screen and remains visible in all workspaces. It provides:
   - o Activities button for navigating workspaces and applications
   - o System controls for volume, networking, and calendar
   - o Keyboard layout switcher (if multiple layouts are configured)

2) **Activities Overview**
   This view helps manage open windows and launch applications. Access it by clicking Activities or pressing the Super (Windows) key. It includes:
   - o Dash: A quick-launch bar for favorite and running applications
   - o Windows Overview: A central area displaying all open windows
   - o Workspace Selector: A right-side panel for managing multiple workspaces

3) **System Menu**
   Located at the top-right, this menu lets users:
   - o Adjust screen brightness
   - o Manage network connections
   - o Open Settings
   - o Lock the screen or log out

4) **Dash (Dock)**
   The Dash holds frequently used applications, running apps, and the Show Applications button to browse all installed programs.

5) **Windows Overview**
   Displays previews of open windows for easy navigation and workspace organization.

6) **Workspace Selector**
   Enables seamless switching and window movement between different workspaces.

7) **Message Tray**
   Shows system and app notifications. Click the clock on the top bar or press Super+M to open it. It also displays the calendar and event details.

Keyboard Shortcuts in GNOME

To customize keyboard shortcuts:

1. Open the System Menu (top-right corner).
2. Click Settings → Keyboard → Keyboard Shortcuts.

Note: Some shortcuts (e.g., function keys, Super key) may not work inside a virtual machine, as they might be captured by your local OS. To send them to a VM, use the on-screen keyboard by clicking the keyboard icon at the top of the VM interface.

Access Workspaces/Using Workspaces in GNOME

Workspaces help organize application windows. For example, system admin tools can be kept in one workspace, while communication apps (email, chat) remain in another.

Switching Workspaces:

1. Press Ctrl+Alt+LeftArrow or Ctrl+Alt+RightArrow to move between workspaces.
2. Open Activities Overview and select a workspace.

Moving Windows Between Workspaces:

- Drag windows between workspaces using the **Workspace Selector**.

Start a Terminal/Launching a Terminal in GNOME

To open a terminal window for a shell prompt:

- Click Activities → Terminal in the Dash or search for "Terminal."
- Use the Show Applications button to find it.
- Press Alt+F2, type gnome-terminal, and hit Enter.

The terminal prompt displays the **current user**, **hostname**, and **working directory**.

Locking the Screen and Logging Out

To lock the screen, either:

- Click the Lock button in the System Menu (top-right).
- Press Super+L (Windows+L).

To unlock, press Enter or click the mouse, then enter your password.
To log out, open the System Menu → Power Off/Log Out → Log Out.

Power Off or Reboot the System - Shutting Down or Restarting

To shut down the system:

- Open the System Menu (top-right).
- Click Power Off/Log Out → Power Off.

or press Ctrl+Alt+Del. A window is displayed that offers the option to Cancel or confirm the Power Off action. If you do not make a choice, then the system automatically shuts down after 60 seconds.

To restart, choose Restart instead. If no action is taken, the system will shut down or restart automatically in 60 seconds.

Further Reading and Resources

- **GNOME Help:** Run yelp in the terminal
- **GNOME Overview Guide:** yelp help:gnome-help/shell-introduction
- **GNOME 40 Web Page:** https://forty.gnome.org/

Execute Commands with the Bash Shell

Objectives

Utilize Bash shortcuts to streamline command execution at the shell prompt.

Basic Command Structure

Bash (GNU Bourne-Again Shell) is a command-line interpreter that processes user-entered commands. A command entered into the shell typically consists of three components:

1. Command – The name of an installed program.
2. Options – Modifiers that usually begin with a hyphen (-) or double hyphen (--).
3. Arguments – Additional inputs for the command.

Each part of a command is separated by spaces. Once a command is entered, pressing Enter executes it. The output is displayed before the next shell prompt appears.

```
[user@host ~]$ whoami
user
[user@host ~]$
```

To execute multiple commands on a single line, use a semicolon (;), a special character in Bash known as a metacharacter.

```
[user@host ~]$ command1 ; command2
command1 output
command2 output
[user@host ~]$
```

Executing Basic Commands

The **date** command prints the system’s current date and time. Privileged users can modify the system clock using this command.

```
[user@host ~]$ date
Sun Feb 27 08:32:42 PM EST 2022
[user@host ~]$ date +%R
20:33
[user@host ~]$ date +%x
02/27/2022
```

The **passwd** command updates a user's password. If no options are specified, it modifies the password of the current user. The system enforces password complexity, requiring a mix of lowercase, uppercase, numbers, and special characters.

```
[user@host ~]$ passwd
Changing password for user user.
Current password: old_password
New password: new_password
Retype new password: new_password
passwd: all authentication tokens updated successfully.
```

Linux does not rely on file extensions for classification. Instead, the **file** command determines file types by inspecting their headers.

```
[user@host ~]$ file /etc/passwd
/etc/passwd: ASCII text
[user@host ~]$ file /bin/passwd
/bin/passwd: setuid ELF 64-bit executable, x86-64, dynamically linked
```

Viewing File Contents

The cat command displays file contents, concatenates multiple files, and redirects output.

```
[user@host ~]$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
```

To view multiple files:

```
[user@host ~]$ cat file1 file2
Hello World!!
Introduction to Linux commands.
```

For large files, **less** provides paginated viewing, allowing scrolling with **UpArrow** and **DownArrow**. Press **q** to exit.

The **head** and **tail** commands display the first and last ten lines of a file, respectively. Use the **-n** option to customize the number of lines.

```
[user@host ~]$ head /etc/passwd
[user@host ~]$ tail -n 3 /etc/passwd
```

The **wc** command counts lines, words, and characters in a file.

```
[user@host ~]$ wc -l /etc/passwd ; wc -l /etc/group
```

Using Tab Completion

Tab completion speeds up command and filename entry. If a typed string is ambiguous, pressing Tab twice displays all matching commands.

```
[user@host ~]$ pasTab+Tab
passwd paste pasuspender
```

Tab completion also works for filenames:

```
[user@host ~]$ ls /etc/pasTab
passwd passwd-
```

It assists with long option names for commands like **useradd**:

```
[root@host ~]# useradd --Tab+Tab
--badnames --gid --shell --home-dir --password --uid
```

Executing Long Commands Over Multiple Lines

To improve readability, use the backslash **(\)** or **the escape character** to break long commands into multiple lines.

```
[user@host ~]$ head -n 3 \
/usr/share/dict/words \
/usr/share/dict/linux.words
```

Bash indicates continuation with the > secondary prompt, but users should avoid typing this manually, as it is also used for output redirection.

Using Command History

The history command lists previously executed commands, prefixed with a number.

```
[user@host ~]$ history
23 clear
24 who
25 pwd
26 ls /etc
27 uptime
```

Reuse past commands with:

- **!number** – Re-executes a command by its history number.
- **!string** – Runs the most recent command starting with the specified string.

```
[user@host ~]$ !26
ls /etc
```

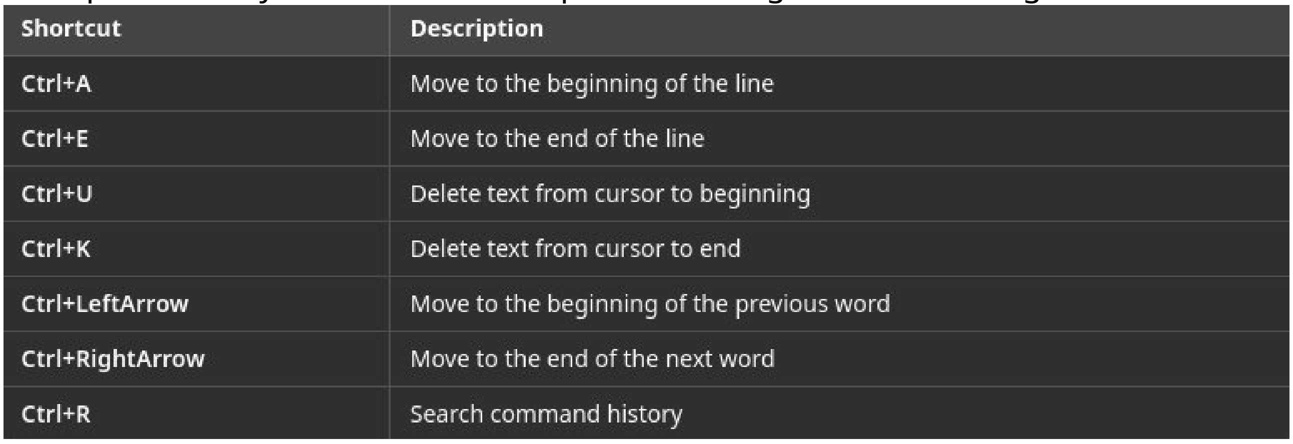
Arrow keys navigate history:

- **UpArrow** – Moves to the previous command.
- **DownArrow** – Moves to the next command.
- **LeftArrow/RightArrow** – Moves within the current command for editing.

Use Esc+. or Alt+. to insert the last word from the previous command.

Editing Commands Efficiently

Bash provides key combinations for quick text navigation and editing:



Shortcut	Description
Ctrl+A	Move to the beginning of the line
Ctrl+E	Move to the end of the line
Ctrl+U	Delete text from cursor to beginning
Ctrl+K	Delete text from cursor to end
Ctrl+LeftArrow	Move to the beginning of the previous word
Ctrl+RightArrow	Move to the end of the next word
Ctrl+R	Search command history

For further details, consult relevant man pages:

```
man bash
man date
man file
man passwd
man cat
man less
man head
man tail
man wc
```

By mastering these commands and shortcuts, users can significantly improve their efficiency in Bash.

Enhance Command-line Efficiency & Bash Scripting Skills

Learning Objectives

- Master advanced Bash shell features and commands to execute tasks more effectively.
- Develop and execute Bash scripts to automate system administration tasks.
- Utilize essential Red Hat Enterprise Linux utilities for streamlined operations.

Writing and Running Bash Scripts

System administrators often rely on command-line tools for task execution. While simple tasks can be performed with individual commands, complex workflows necessitate chaining multiple commands together. By leveraging Bash scripting, you can create automated solutions for repetitive tasks, improving both efficiency and accuracy.

A Bash script is an executable file containing a sequence of commands, potentially with logic structures for decision-making. When written efficiently, a script can function as a standalone command-line utility and be incorporated into broader automation workflows.

Shell scripting proficiency is invaluable in operational environments, enabling administrators to execute routine tasks consistently and efficiently. While any text editor can be used for script creation, advanced editors like Vim or Emacs provide syntax highlighting and error detection, helping prevent common mistakes such as mismatched quotes or incorrect syntax.

Specifying the Command Interpreter

The first line of any Bash script should start with a shebang **(#!)**, a notation used to specify the interpreter that will process the script. In Bash scripting, the following directive is commonly used:

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

This ensures that the system correctly executes the script using the specified shell interpreter.

Executing a Bash Script

To run a script, it must have execute permissions. The **chmod** command is used to modify file permissions, while **chown** can grant execute access to specific users or groups.
If the script is stored in a directory included in the system's **PATH** variable, you can execute it simply by its filename. Otherwise, you must provide its full path or use the **./** prefix if running it from the current directory.

```
[user@host ~]$ which myscript
~/bin/myscript
[user@host ~]$ echo $PATH
/home/user/.local/bin:/home/user/bin:/sbin:/bin:/usr/sbin:/usr/bin:/u
sr/local/sbin:/usr/local/bin
```

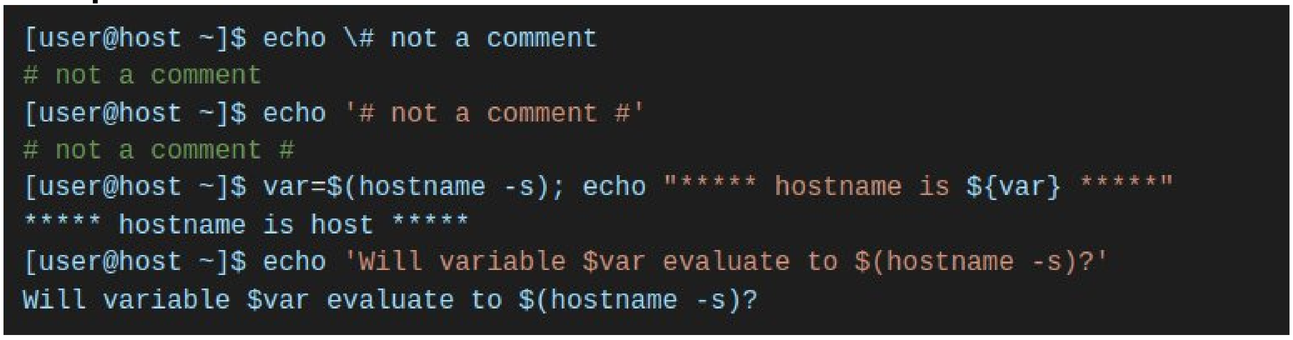
To avoid conflicts, never name a script using an existing system command.

Handling Special Characters in Bash

Bash interprets certain characters and words with special meanings. To use them as literals, they must be escaped using:

- Backslashes **(\)** for individual characters.
- Single quotes **(' ')** to preserve all enclosed characters literally.
- Double quotes **(" ")** to suppress globbing and shell expansion while allowing variable substitution.

Examples:



```
[user@host ~]$ echo \# not a comment
# not a comment
[user@host ~]$ echo '# not a comment #'
# not a comment #
[user@host ~]$ var=$(hostname -s); echo "***** hostname is ${var} *****"
***** hostname is host *****
[user@host ~]$ echo 'Will variable $var evaluate to $(hostname -s)?'
Will variable $var evaluate to $(hostname -s)?
```

Understanding proper quoting and escaping techniques prevents unintended behavior in scripts.

Displaying Output from a Script

The **echo** command is a fundamental tool for displaying output in a Bash script. By default, it prints text to **standard output (STDOUT)**, but its output can be redirected as needed.

Example script:

```
[user@host ~]$ cat ~/bin/hello
#!/usr/bin/bash
echo "Hello, world"
```

If **~/bin/** is in the **PATH**, running **hello** will execute the script.

Redirecting Output

For structured logging and debugging, messages should be appropriately categorized:

- Standard Output (STDOUT) for general information.
- Standard Error (STDERR) for error messages.
- Redirection (> and >>) to store messages in files.

```
[user@host ~]$ cat ~/bin/hello
#!/usr/bin/bash
echo "Hello, world"
echo "ERROR: Houston, we have a problem." >&2
[user@host ~]$ hello 2> hello.log
[user@host ~]$ cat hello.log
ERROR: Houston, we have a problem.
```

The > and >> operators are used for redirecting output to a file:

- **> (single greater-than)**
  - If the file does not exist, it creates the file and writes to it.
  - If the file exists, it overwrites the file, deleting any existing content.

- **>> (double greater-than)**
  - If the file does not exist, it creates the file and writes to it.
  - If the file exists, it appends the new content to the file instead of overwriting it.

Debugging with echo

Adding **echo** statements in scripts helps track variable values and execution flow, aiding in troubleshooting issues efficiently.

References

For more in-depth details, consult the bash(1), echo(1), and echo(1p) man pages.

Loops and Conditional Constructs in Scripts

Objectives

Mastering loops and conditional constructs is essential for automating repetitive tasks in scripting. This guide covers using **for** loops for iteration, evaluating exit codes from commands and scripts, performing logical tests with operators, and implementing conditional structures using **if** statements.

Utilizing Loops for Command Iteration

System administrators frequently perform repetitive operations. Examples include:

- Monitoring a process every minute for ten minutes to check its completion status.
- Executing a command sequentially on multiple targets, such as backing up numerous databases.

The **for** loop in Bash facilitates these iterations efficiently.

Processing Items from the Command Line

The **for** loop follows this syntax:

```
for VARIABLE in LIST; do
    COMMAND VARIABLE
done
```

Here, **VARIABLE** takes each value from **LIST**, and the command block executes for each instance. The list values can be user-defined, generated via shell expansion, or derived from command substitution.

Examples of Providing Lists to for Loops

```
[user@host ~]$ for HOST in host1 host2 host3; do echo $HOST; done
host1
host2
host3

[user@host ~]$ for HOST in host{1,2,3}; do echo $HOST; done
host1
host2
host3

[user@host ~]$ for HOST in host{1..3}; do echo $HOST; done
host1
host2
host3

[user@host ~]$ for FILE in file{a..c}; do ls $FILE; done
filea
fileb
filec

[user@host ~]$ for PACKAGE in $(rpm -qa | grep kernel); \
do echo "$PACKAGE was installed on \
$(date -d @$(rpm -q --qf "%{INSTALLTIME}\n" $PACKAGE))"; done
```

Exit Codes in Bash Scripts

Every command in a script returns an exit code upon execution:

- **0** indicates successful execution.
- **Nonzero** values denote errors, allowing differentiation between failure types.

Using the exit Command

A script can terminate at any point using:

```
exit N
```

where **N** is an integer from **0** to **255**, signifying the exit status.

Example: Checking Exit Codes

```
[user@host bin]$ cat hello
#!/usr/bin/bash
echo "Hello, world"
exit 0

[user@host bin]$ ./hello
Hello, world
[user@host bin]$ echo $?
0
```

If **exit** is omitted, the script returns the last command's exit code.

Logical Testing in Bash Scripts

To ensure scripts handle unexpected conditions, verify inputs such as user arguments, variable expansions, and command substitutions using the **test** command.
To see the exit status, view the $? variable immediately after executing the test command.

```
[user@host ~]$ test 1 -gt 0 ; echo $?
0
[user@host ~]$ test 0 -gt 1 ; echo $?
1
```

Numeric Comparison Operators

```
[user@host ~]$ [[ 8 -gt 2 ]]; echo $?
0
[user@host ~]$ [[ 2 -lt 2 ]]; echo $?
1
```

Test by using the Bash test command syntax, [<TESTEXPRESSION>] or the newer extended test command syntax, [[<TESTEXPRESSION>]], which provides features such as file name globbing and regex pattern matching. In most cases, use the [[<TESTEXPRESSION>]] syntax.
The space characters inside the brackets are mandatory, because they separate the words and elements within the test expression.

String Comparison Operators & Testing String Lengths

```
[user@host ~]$ [[ abc = abc ]]; echo $?
0
[user@host ~]$ [[ abc == def ]]; echo $?
1
[user@host ~]$ [[ abc != def ]]; echo $?
0

[user@host ~]$ STRING=''; [[ -z "$STRING" ]]; echo $?
0
[user@host ~]$ STRING='abc'; [[ -n "$STRING" ]]; echo $?
0
```

- **-z string:** True if the length of string is zero.
- **-n string:** True if the length of string is non-zero.

File and Directory Tests

- **-f** checks if a file exists.
- **-d** verifies if a directory exists.
- **-L** checks for symbolic links.
- **-r** ensures the user has read permissions.

Conditional Constructs in Bash

Scripts often need decision-making logic to execute specific commands based on conditions. The **if** statement enables this functionality.

Basic if/then Construct

```
if <CONDITION>; then
    <STATEMENT>
fi
```

Example:

```
[user@host ~]$ systemctl is-active psacct > /dev/null 2>&1
[user@host ~]$ if [[ $? -ne 0 ]]; then sudo systemctl start psacct; fi
```

Extending with if/then/else

When a script must execute different commands based on conditions:

```
if <CONDITION>; then
    <STATEMENT>
else
    <STATEMENT>
fi
```

Example:

```
[user@host ~]$ systemctl is-active psacct > /dev/null 2>&1
[user@host ~]$ if [[ $? -ne 0 ]]; then \
sudo systemctl start psacct; \
else \
sudo systemctl stop psacct; \
fi
```

Multiple Conditions with if/then/elif/else

To handle multiple conditions in sequence:

```
if <CONDITION>; then
    <STATEMENTS>
elif <CONDITION>; then
    <STATEMENTS>
else
    <STATEMENTS>
fi
```

Example:

```
[user@host ~]$ systemctl is-active mariadb > /dev/null 2>&1
[user@host ~]$ MARIADB_ACTIVE=$?
[user@host ~]$ systemctl is-active postgresql > /dev/null 2>&1
[user@host ~]$ POSTGRESQL_ACTIVE=$?
[user@host ~]$ if [[ "$MARIADB_ACTIVE" -eq 0 ]]; then \
mysql; \
elif [[ "$POSTGRESQL_ACTIVE" -eq 0 ]]; then \
psql; \
else \
sqlite3; \
fi
```

References

For further details, consult the following resources:

- bash(1) man page
- test(1) man page
- getopt(3) man page

Matching Text in Command Output Using Regular Expressions

Objectives

Learn to construct regular expressions for pattern matching, apply them to text files using the **grep** command, and leverage **grep** for searching within files and piped command outputs.

Understanding Regular Expressions

Regular expressions offer a robust method for identifying specific patterns in text. Tools such as **vim**, **grep**, and **less** support regular expressions. Additionally, programming languages like **Perl**, **Python**, and **C** implement regular expressions with slight syntax variations.
Regular expressions function as a distinct language with their own set of rules and syntax. The following sections will explore how they are used in **bash**, with practical examples.

Simple Regular Expressions

The most fundamental form of a regular expression is an exact match. An exact match occurs when the characters in the pattern align precisely with the target string, including order and type.

For instance, if a user searches for the term **cat** in the following file:

```
cat
dog
concatenate
dogma
category
educated
boondoggle
vindication
chilidog
```

The pattern **cat** will match instances where **c, a, and t** appear sequentially without interruption. Running **grep 'cat' filename** would yield:

```
cat
concatenate
category
educated
vindication
```

Matching the Beginning and End of a Line

By default, a regular expression finds matches anywhere within a line. However, line anchors allow us to specify matches at the start or end of a line.

- The caret **(^)** ensures that a match occurs at the beginning of a line.
- The dollar sign **($)** ensures that a match occurs at the end of a line.

For example:

- The pattern **^cat** matches:
  ```
  cat
  category
  ```
- The pattern **cat$** matches:
  ```
  cat
  ```
- The pattern **dog$** locates lines ending in dog:
  ```
  dog
  chilidog
  ```
- To match cat as an entire line, use **^cat$**, which matches:
  ```
  cat
  ```

Basic & Extended Regular Expressions

Regular expressions fall into two categories: basic and extended.

- Basic regular expressions require special characters like **|, +, ?, (), {, and }** to be prefixed with a backslash **(\)**.
- Extended regular expressions interpret these characters as special unless prefixed with \.

Commands like **grep**, **sed**, and **vim** default to basic expressions. Using **grep -E** or **sed -E** enables extended expressions.

Wildcards and Multipliers in Regular Expressions

The dot **(.)** serves as a wildcard that matches any single character. For example, **c.t** matches words like **cat**, **cut**, and **c$t**.
To restrict matches to specific characters, use brackets. For instance, **c[aou]t** matches **cat**, **cot**, and **cut**.

Multipliers extend pattern flexibility:

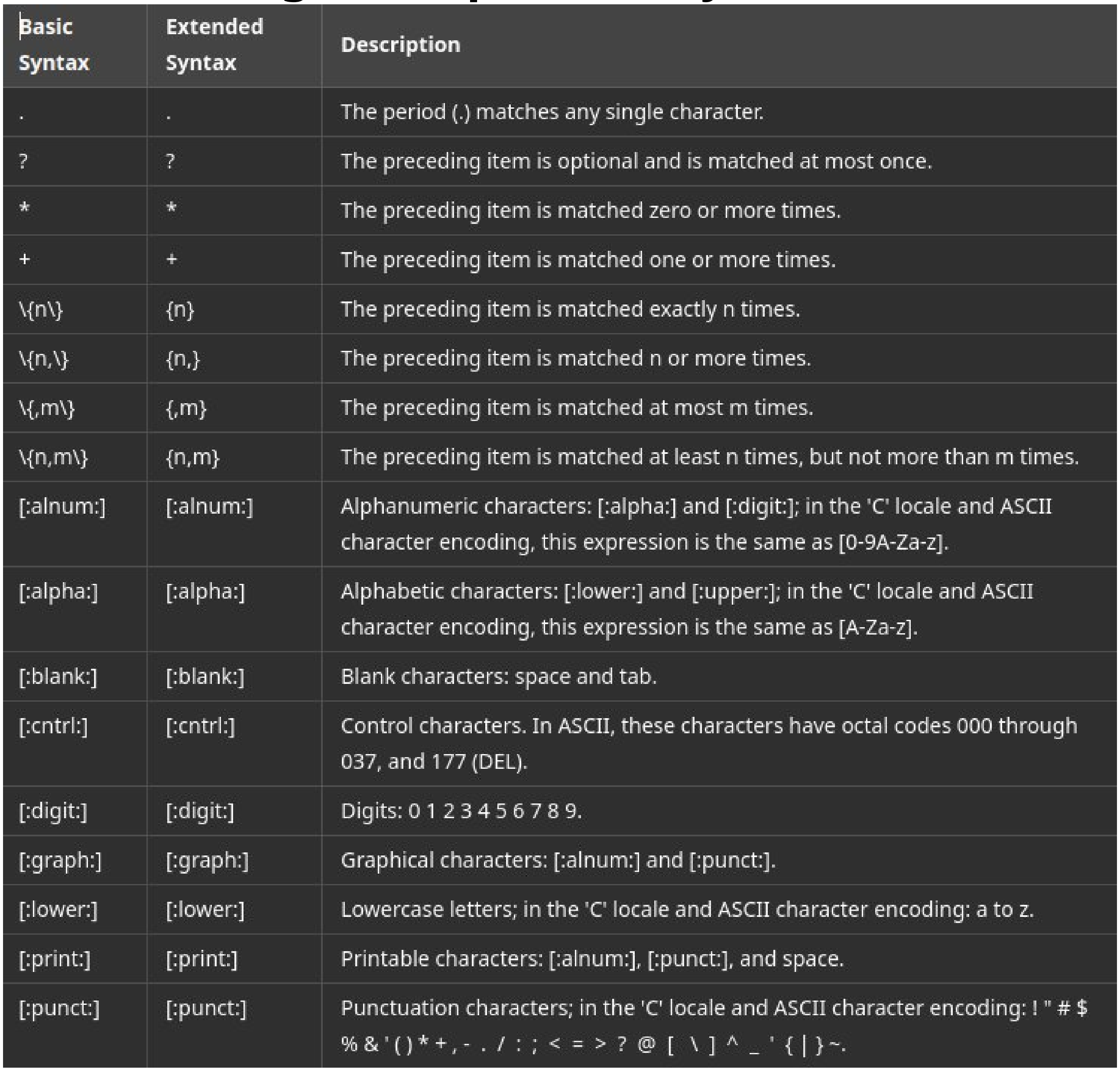
- The asterisk **(*)** matches zero or more occurrences of the preceding character.
    - **c[aou]*t** could match **coat** or **coot**.
    - **c.*t** matches **cat**, **coat**, **culvert**, or even **ct**.
- Explicit multipliers define precise repetition:
    - **c.\{2\}t** matches words with exactly two characters between **c** and **t**, such as **cat** and **coat**.

Regular Expressions vs. Shell Pattern Matching

Shell pattern matching (globbing) and regular expressions both use metacharacters, like *****, but differ significantly:

- Shell patterns match filenames.
- Regular expressions match text patterns in command outputs and files.

Common Regular Expression Syntax



Basic Syntax	Extended Syntax	Description
.	.	The period (.) matches any single character.
?	?	The preceding item is optional and is matched at most once.
*	*	The preceding item is matched zero or more times.
+	+	The preceding item is matched one or more times.
\{n\}	{n}	The preceding item is matched exactly n times.
\{n,\}	{n,}	The preceding item is matched n or more times.
\{,m\}	{,m}	The preceding item is matched at most m times.
\{n,m\}	{n,m}	The preceding item is matched at least n times, but not more than m times.
[:alnum:]	[:alnum:]	Alphanumeric characters: [:alpha:] and [:digit:]; in the 'C' locale and ASCII character encoding, this expression is the same as [0-9A-Za-z].
[:alpha:]	[:alpha:]	Alphabetic characters: [:lower:] and [:upper:]; in the 'C' locale and ASCII character encoding, this expression is the same as [A-Za-z].
[:blank:]	[:blank:]	Blank characters: space and tab.
[:cntrl:]	[:cntrl:]	Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL).
[:digit:]	[:digit:]	Digits: 0 1 2 3 4 5 6 7 8 9.
[:graph:]	[:graph:]	Graphical characters: [:alnum:] and [:punct:].
[:lower:]	[:lower:]	Lowercase letters; in the 'C' locale and ASCII character encoding: a to z.
[:print:]	[:print:]	Printable characters: [:alnum:], [:punct:], and space.
[:punct:]	[:punct:]	Punctuation characters; in the 'C' locale and ASCII character encoding: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ' { | } ~.

[:space:]	[:space:]	Space characters: in the 'C' locale, it is tab, newline, vertical tab, form feed, carriage return, and space.
[:upper:]	[:upper:]	Uppercase letters: in the 'C' locale and ASCII character encoding, it is: A to Z.
[:xdigit:]	[:xdigit:]	Hexadecimal digits: 0-9, A-F, a-f.
\b	\b	Match the empty string at the edge of a word.
\B	\B	Match the empty string provided that it is not at the edge of a word.
\<	\<	Match the empty string at the beginning of a word.
\>	\>	Match the empty string at the end of a word.
\w	\w	Match word constituent. Synonym for [_[:alnum:]].
\W	\W	Match non-word constituent. Synonym for [^_[:alnum:]].
\s	\s	Match white space. Synonym for [[:space:]].
\S	\S	Match non-white space. Synonym for [^[:space:]].

Using Regular Expressions with grep

The **grep** command searches files for lines that match a specified pattern.

Example: Searching a Dictionary

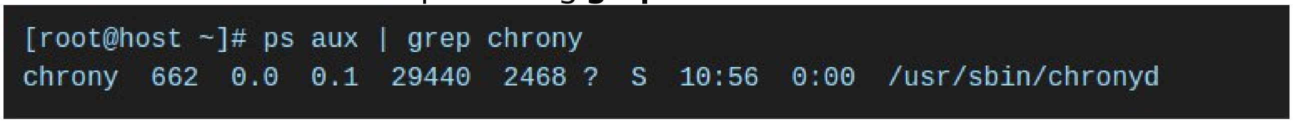
```
[user@host ~]$ grep '^computer' /usr/share/dict/words
computer
computerese
computerise
computerite
computerizable
computerization
computerize
computerized
computerizes
computerizing
computerlike
computernik
computers
```

Important Note

Always enclose patterns in single quotes to prevent shell interpretation of metachar-acters, e.g., **grep '^pattern$' file.txt**.

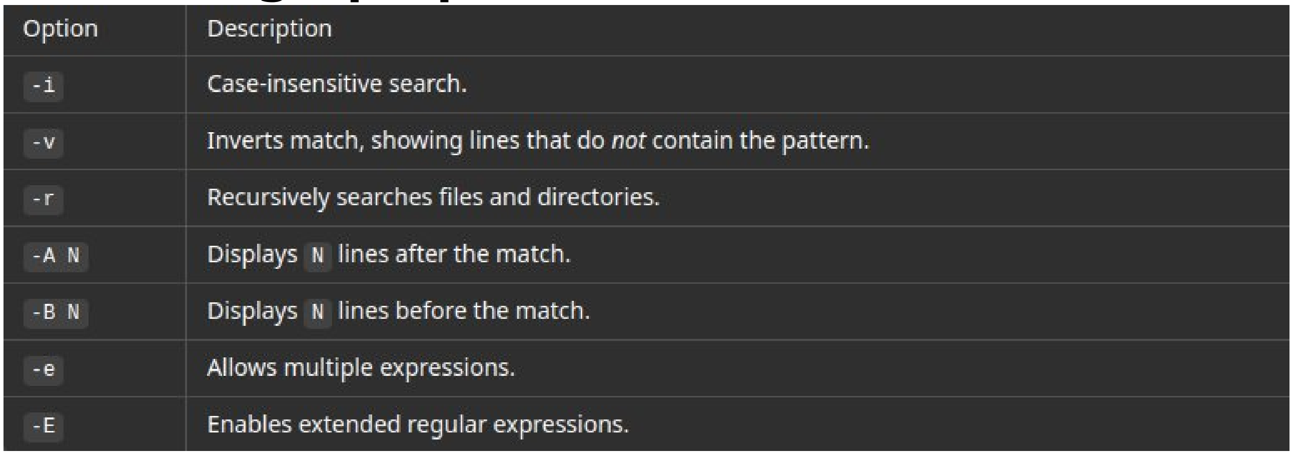
Using Pipes with grep

You can filter command outputs using **grep**:



```
[root@host ~]# ps aux | grep chrony
chrony  662  0.0  0.1  29440  2468 ?  S  10:56  0:00  /usr/sbin/chronyd
```

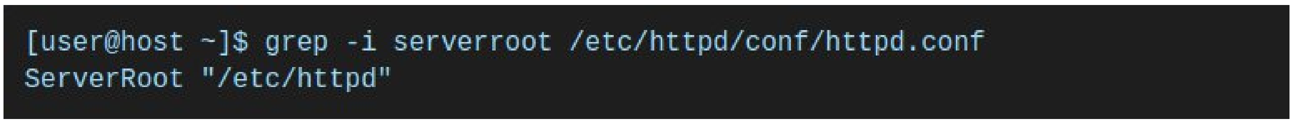
Common grep Options



Option	Description
`-i`	Case-insensitive search.
`-v`	Inverts match, showing lines that do *not* contain the pattern.
`-r`	Recursively searches files and directories.
`-A N`	Displays `N` lines after the match.
`-B N`	Displays `N` lines before the match.
`-e`	Allows multiple expressions.
`-E`	Enables extended regular expressions.

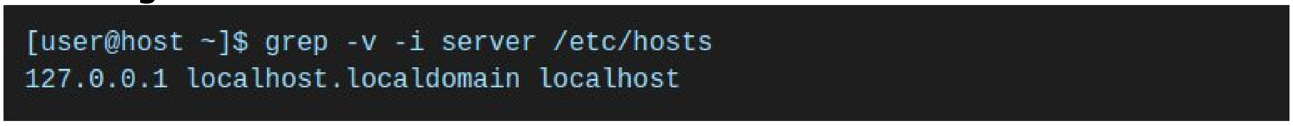
Practical Examples

Case-Insensitive Search



```
[user@host ~]$ grep -i serverroot /etc/httpd/conf/httpd.conf
ServerRoot "/etc/httpd"
```

Excluding Lines



```
[user@host ~]$ grep -v -i server /etc/hosts
127.0.0.1 localhost.localdomain localhost
```

Filtering Comments

`[user@host ~]$ grep -v '^[#;]' /etc/systemd/system/multi-user.target.wants/rsys`

Searching for Multiple Patterns

`[root@host ~]# cat /var/log/secure | grep -e 'pam_unix' -e 'user root' -e 'Accepted publickey' | less`

References

regex(7) and grep(1) man pages

Conclusion

Regular expressions provide a powerful tool for searching and manipulating text. By mastering them alongside **grep**, you can efficiently parse files and command outputs, making your workflow more productive and efficient.