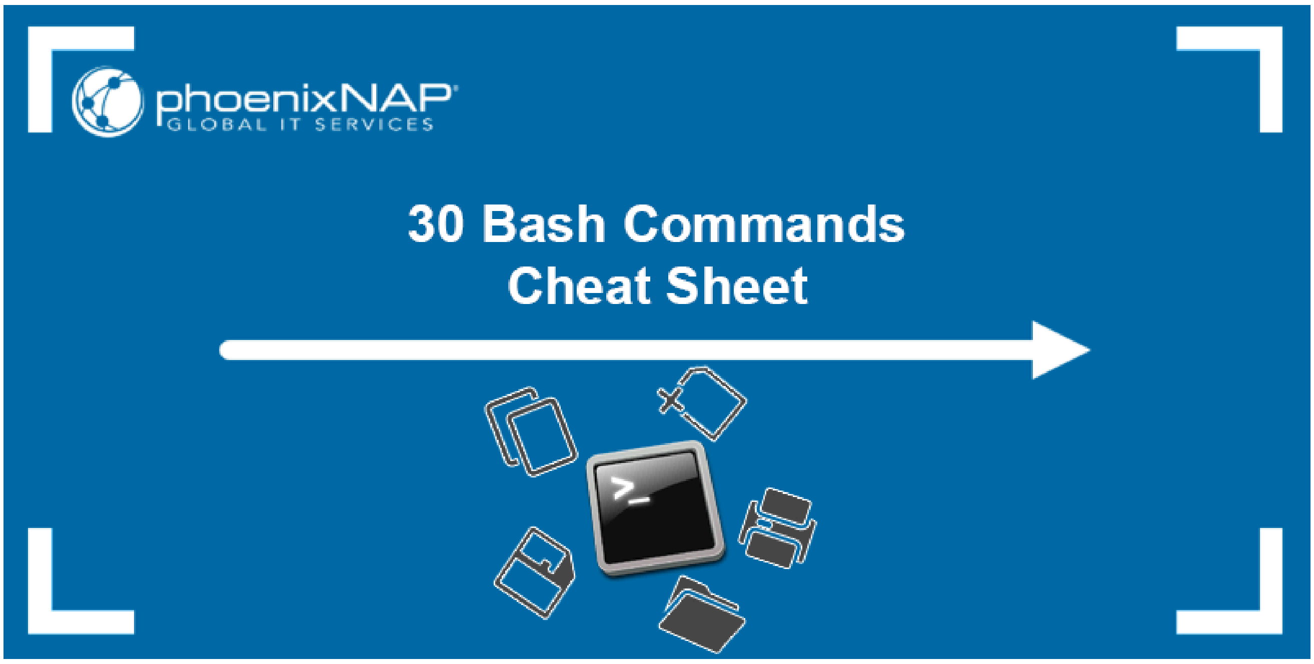
Introduction

The Bash (Bourne Again Shell) shell is an enhanced version of the Bourne shell distributed with Linux and GNU operating systems. Basic Bash commands allow users to navigate through a system and effectively manage files, directories, and different data types.

This article will list 30 Bash commands and provide you with a downloadable PDF cheat sheet to always have them at hand.



Note: Check out our guide to Linux shells.

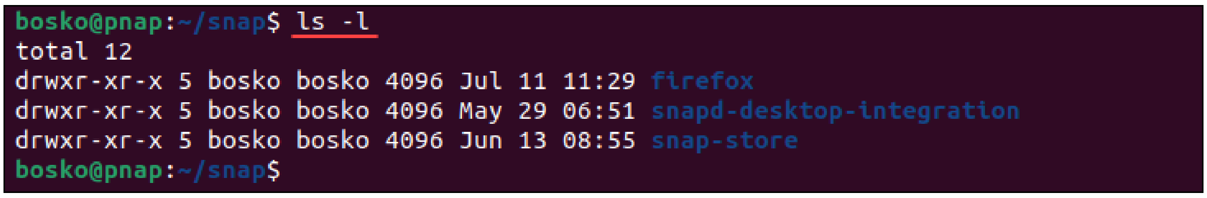
The syntax of the **ls** command is:

```
ls [options] [file|dir]
```

The most commonly used options are:

Option	Description
`-l`	Provide detailed information about files and directories.
`-a`	By default, **ls** omits hidden files from the output. To include hidden files, specify the **-a** option.
`-h`	Modifies the output to display file sizes in a more human-friendly format. Instead of using bytes, the **-h** option formats the sizes using units like kilobytes (KB), megabytes (MB), gigabytes (GB), etc.

The following example shows the output of the **ls -l** command:



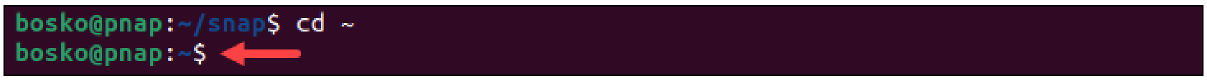
```
bosko@pnap:~/snap$ ls -l
total 12
drwxr-xr-x 5 bosko bosko 4096 Jul 11 11:29 firefox
drwxr-xr-x 5 bosko bosko 4096 May 29 06:51 snapd-desktop-integration
drwxr-xr-x 5 bosko bosko 4096 Jun 13 08:55 snap-store
bosko@pnap:~/snap$
```

cd Command

The cd command changes the current directory to the directory specified as an argument to the command. It allows users to move to different locations in the file system and access files, directories, and resources in those directories.

The command is essential for efficient command-line navigation and for executing commands or accessing files in specific directories without having to provide full paths each time.

The syntax for the **cd** command is:


```
bosko@pnap:~/snap$ cd ~
bosko@pnap:~$
```

pwd Command

The pwd command prints the path of the current working directory. It is useful when you need to find your way within the Linux file system or to pass the working directory in a Bash script.

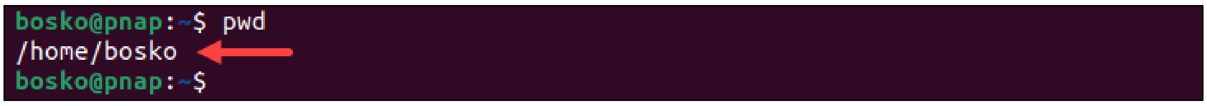
The syntax for the **pwd** command is:

```
pwd [options]
```

The most commonly used options are:

Option	Description
-L	Prints the logical path of the current directory.
-P	Prints the physical path of the current directory.

The following example shows the output of the **pwd** command:


```
bosko@pnap:~$ pwd
/home/bosko
bosko@pnap:~$
```

File Manipulation

Use the commands listed in this section for file manipulation, including file creation, copying, moving, renaming, deleting, searching, modifying files, etc.

cat Command

The cat command is short for concatenate. It displays the contents of a file or files in the terminal without having to open the file for editing. It is convenient for viewing the contents of small text files, joining together and displaying multiple files' contents, or piping the output into other commands for further processing.

The syntax for the **cat** command is:

```
cat [options] [file path]
```

touch Command

The touch command creates a new, empty file or updates the timestamp of an existing file. The command is useful when you want to create a new file without any content or when you need to modify the timestamps of a file to the current time without changing its contents.

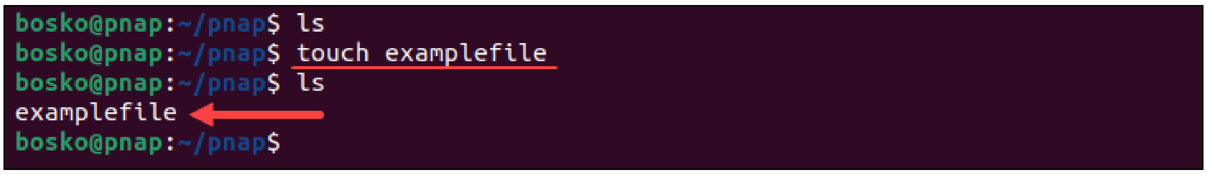
The syntax of the **touch** command is:

```
touch [file]
```

The most commonly used options are:

Option	Description
`-a`	Changes access time only.
`-m`	Changes modification time only.

The following example shows how to create a new blank file using the **touch** command:



```
bosko@pnap:~/pnap$ ls
bosko@pnap:~/pnap$ touch examplefile
bosko@pnap:~/pnap$ ls
examplefile
bosko@pnap:~/pnap$
```

rm Command

The rm command removes files or directories to free up storage space and organize the file system.

Important: Use **rm** with caution, as deleted files are not recoverable by default, and unintended removals can result in data loss. The command deletes files without prompting for confirmation. Check out our article for other dangerous Linux commands.

The syntax for the **rm** command is:

```
rm [options] [file|dir path]
```

	confirmation.

mkdir Command

The mkdir command creates a new, empty directory. **`mkdir`** also allows users to set permissions, create multiple directories at once, and much more. The tool is handy when setting up a new project structure or when you need to create nested directories to maintain a well-organized file system.

The syntax of the **`mkdir`** command is:

```
mkdir [options] [dir]
```

The commonly used options that provide additional functionality are:

Option	Description
`-p`	A necessary option if you want to create subdirectories as well. Without the **`-p`** option, the output is an error if one of the directories in the string does not exist.
`-m [mode]`	Sets the permissions (mode) for the newly created directory. For **`[mode]`**, specify the permission mode using the octal representation, like 777, to add read, write, and execute permission for all users.

rmdir Command

The rmdir command removes only empty directories via the terminal. It is useful when you need to clean up the file system by removing empty directories that are no longer needed, which helps maintain a tidy directory structure.

The syntax for the **`rmdir`** command is:

```
rmdir [options] [dir]
```

attributes.

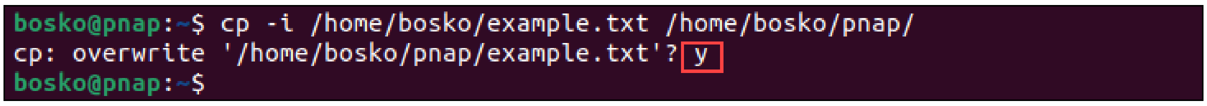
The **cp** command is essential when creating backups, replicating data, or organizing files across different directories. The syntax is:

```
cp [options] [source] [destination]
```

The commonly used options are:

Option	Description
`-r`	Enforces recursive mode. It allows the **cp** command to copy the specified directory and its entire contents, including subdirectories and their contents.
`-i`	Enforces the interactive mode, prompting the user for confirmation before overwriting any existing files.
`-n`	The **-n** option stands for *no clobber* and prevents the **cp** command from overwriting existing files.

The following example shows how the interactive mode prompts the user for confirmation before overwriting an existing file:



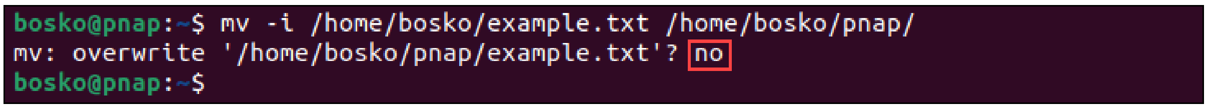
```
bosko@pnap:~$ cp -i /home/bosko/example.txt /home/bosko/pnap/
cp: overwrite '/home/bosko/pnap/example.txt'? y
bosko@pnap:~$
```

mv Command

The mv command allows users to move directories and files within the same file system, effectively changing their path. Additionally, it can also rename files or directories, providing a way to organize and manage data within the file system efficiently.

The syntax for the **mv** command is:

```
mv [options] [source] [destination]
```



```
bosko@pnap:~$ mv -i /home/bosko/example.txt /home/bosko/pnap/
mv: overwrite '/home/bosko/pnap/example.txt'? no
bosko@pnap:~$
```

Searching and Sorting

The commands listed in this section are useful when searching for files or data in a file system and sorting the results in a specific way.

find Command

The find command searches for files or a specific string of characters in a directory hierarchy. It allows users to search the entire file system or specified directories for files or strings based on factors such as name patterns, file types, sizes, modification times, and more.

Note: Learn how to determine the type of a file using the Linux file command.

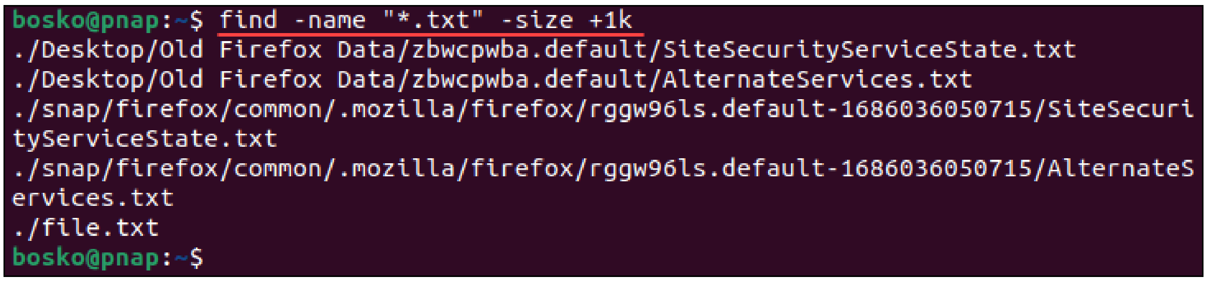
The syntax of the **find** command is:

```
find [location] [expression] [options]
```

- **[location]** is the directory where the search will begin. If not specified, the search starts from the current directory.
- **[expression]** is used to combine multiple search criteria using logical operators (**-and**, **-or**, **-not**). It can be used to create complex search conditions.

The most common command options are:

Option	Description
`-name [pattern]`	Used to search for files and directories based on their names. Provide a pattern or a complete name as an argument to the **-name** option, and the **find** command locates and displays the files or directories that match the specified name.



```
bosko@pnap:~$ find -name "*.txt" -size +1k
./Desktop/Old Firefox Data/zbwcpwba.default/SiteSecurityServiceState.txt
./Desktop/Old Firefox Data/zbwcpwba.default/AlternateServices.txt
./snap/firefox/common/.mozilla/firefox/rggw96ls.default-1686036050715/SiteSecuri
tyServiceState.txt
./snap/firefox/common/.mozilla/firefox/rggw96ls.default-1686036050715/AlternateS
ervices.txt
./file.txt
bosko@pnap:~$
```

grep Command

The grep command searches files for lines that match a given regular expression. It is particularly valuable when analyzing log files, code files, or any text-based data since it helps quickly identify relevant information and extract useful insights from large amounts of text.

The command syntax is:

```
grep [options] [search pattern] [file]
```

- `[search pattern]` is the text or regular expression you're searching for. It can be a simple string or a more complex regular expression.
- `[file]` are the files in which you want to search for the pattern. You can specify one or multiple files. If no files are specified, `grep` reads from standard input.

For additional functionality, use the following options:

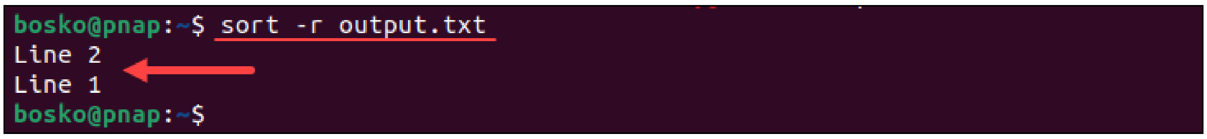
Option	Description
`-i`	Ignores case when performing a search.
`-r`	Enforces a recursive search and searches through directories and their subdirectories. With this option, `grep` searches for the specified pattern in files within the specified directory and all its subdirectories.
`-l`	Displays only the file names that contain the specified pattern instead of showing the matching lines within those files.

The following example shows how to search for a string in a specific file and ignore case:

The most common options are:

Option	Description
`-n`	Used to sort lines of text based on numerical values. When applied, the **`sort`** command interprets the data as numbers and arranges them in ascending order.
`-r`	Used to reverse the sorting order, causing **`sort`** to arrange the lines in descending order instead of ascending.
`-k [argument]`	Used to sort the lines based on a specific field within the lines of text. Provide an argument to the **`-k`** option in the form of **`start_field,end_field`**, where **`start_field`** and **`end_field`** are the fields you want to use for sorting.

In the following example, we use the **`sort`** command to reverse the sorting order in a file in descending order:



```
bosko@pnap:~$ sort -r output.txt
Line 2
Line 1
bosko@pnap:~$
```

Changing Permissions

Changing file permissions in Linux involves modifying the access rights of a file or directory for different user categories (owner, group, and others).

chmod Command

The chmod command modifies the permissions of files and directories. It allows users to specify the access level that the owner, group, and other users have to a file or directory. The command assigns read, write, and execute permissions using symbolic or octal notation. The tool is particularly useful for securing sensitive data, controlling user access to files, and managing script and program execution.

The syntax for the **`chmod`** command is:

```
chmod [options] [mode] [file]
```

permission changes.

Important: Use the **`chmod`** command with caution when dealing with sensitive data as it modifies file permissions and could provide read or write access to users who should not have it.

chown Command

The chown command changes the ownership of files and directories. It allows users with appropriate permissions to reassign the ownership of a file or directory to a different user or group. The command is useful when system administrators need to transfer files between users or change the ownership of system files. This ensures proper access control and data management in the file system.

The **`chown`** command syntax is:

```
chown [options] owner[:group] file
```

- **`owner`** is the name or numeric user ID of the new owner you want to assign to the file(s) or directory.
- **`group`** is the name or numeric group ID of the new group you want to assign to the file(s) or directory. Omitting this part keeps the group unchanged.

The commonly used options are:

Option	Description
`-R`	Recursive mode, applies ownership changes to the specified file or directory but also to all its subdirectories and their contents.
`-v`	The verbose option makes **`chown`** display a message for each file or directory it modifies, showing the changes it makes to the ownership.

combines multiple files and directories into a single file, preserving their structure and permissions. The command helps create backups, distribute and transfer groups of files efficiently.

The syntax for the `tar` command is:

```
tar [options] [archive-file] [file | dir...]
```

- `[archive-file]` is the name of the archive file you want to create, extract from, or manipulate. Omitting this causes `tar` to default to using the standard input/output.
- `[file | dir …]` are the files and/or directories you want to include in the archive. You can specify one or more file/directory names. When creating an archive, list the files and directories you want to include. When extracting, this part is usually omitted, and the command extracts the contents of the archive into the current directory.

The common `tar` options are:

Option	Description
`-c`	Create a new archive.
`-x`	Extract from an existing archive.
`-v`	Verbose mode. Displays a detailed output while `tar` is processing files.
`-f [archive-file]`	Use the `-f` option to specify the name of the archive file you're creating or manipulating. For `[archive-file]`, specify the name of the archive you want to work with.
`-z`	Enable gzip compression when creating or extracting an archive.
`-j`	Enable bzip2 compression when creating or extracting an archive.

For example, the following command creates a new archive named *new-archive* from the file named *file.txt*:

```
gzip [options] [file...]
```

The common **gzip** options are:

Option	Description
-d	Decompress a compressed file. Using this option reverses the compression process performed by **gzip** and restores the original file to its uncompressed state.
-k	Use the **-k** option to retain the original files when performing an operation. It's useful when you want to preserve both the original and processed versions of a file without overwriting the original file.

Note: By default, **gzip** replaces the original files unless the **-k** option is used.

gunzip Command

The **gunzip** command is a utility for decompressing files that have been compressed using the gzip compression algorithm. Use it to restore files to their original uncompressed state. The command is necessary for managing file compression and decompression tasks in Unix-based environments.

The syntax is:

```
gunzip [options] [file...]
```

The default behavior is to replace the original compressed files. If you want to keep the original as well, add the **-k** option.

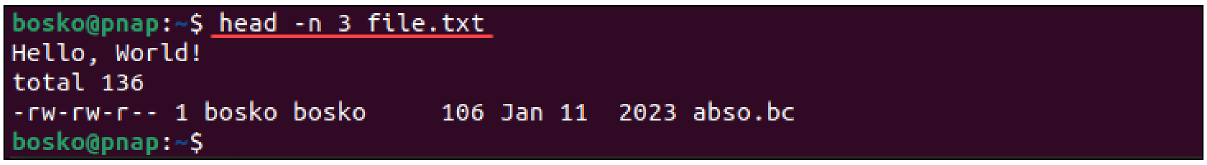
Viewing File and System Details

```
head [options] [file...]
```

The most common options are:

Option	Description
`-n [num]`	Show the first `[num]` number of lines.
`-c [num]`	Shows the first `[num]` bytes of a file.

In the following example, we use the **`head`** command to show the first three lines of a file:



```
bosko@pnap:~$ head -n 3 file.txt
Hello, World!
total 136
-rw-rw-r-- 1 bosko bosko     106 Jan 11  2023 abso.bc
bosko@pnap:~$
```

tail Command

The tail command is a Bash utility used for viewing the ending portion of a text file or input stream. By default, **`tail`** displays the last ten lines of a file, but it can be adjusted to show a specified number of lines or bytes from the end of the file. The command is particularly useful for monitoring log files in real time, tracking changes in files, and obtaining the latest updates without having to examine the entire file.

The **`tail`** command syntax is:

```
tail [options] [file...]
```

The common **`tail`** command options are:

Option	Description
`-n [num]`	Show the last `[num]` number of lines.
`-c [num]`	Shows the last `[num]` bytes of a file.

The following example shows how to use **`tail`** to show the last five lines of a file:

echo allows users to perform basic string manipulation, create prompts, generate user notifications, and create formatted output. The command is essential for both interactive use and automation.

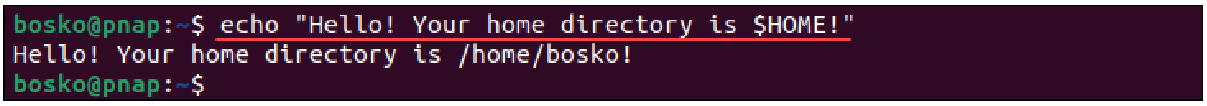
The syntax is:

```
echo [options] [string]
```

The commonly used options are:

Option	Description
`-n`	Instructs **echo** not to print the trailing newline character after displaying the specified text or variable. By default, **echo** adds a newline character at the end of the output.
`-e`	Enables the interpretation of certain escape sequences within the text or variable being echoed.

The following example shows how to use **echo** to print a string and a variable in the standard output:



```
bosko@pnap:~$ echo "Hello! Your home directory is $HOME!"
Hello! Your home directory is /home/bosko!
bosko@pnap:~$
```

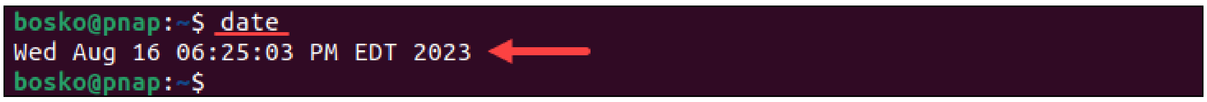
date Command

The date command is a Bash utility that prints the system date and time. It also allows users to format and manipulate date and time information. The **date** command is useful for timestamping files, generating time-sensitive output in scripts, setting system clocks, calculating time intervals, and more.

The **date** command syntax is:

```
date [options] [+format]
```

The following example shows the output of the **date** command:



```
bosko@pnap:~$ date
Wed Aug 16 06:25:03 PM EDT 2023
bosko@pnap:~$
```

df Command

The df command is a Bash utility that allows users to check disk space usage on a filesystem basis. When invoked without arguments, **df** provides a summary of disk space usage across all mounted filesystems.

The command syntax is:

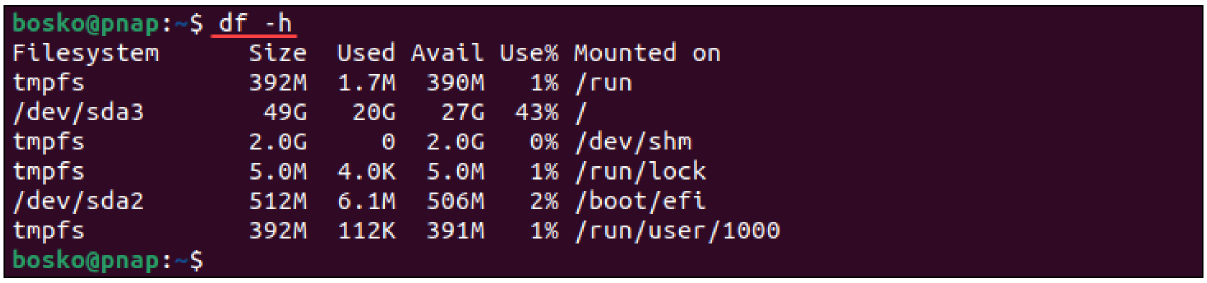
```
df [options] [filesystem]
```

[filesystem] is an optional argument that allows you to specify filesystems. **df** then displays information only for those filesystems. Omitting these arguments causes **df** to display information for all mounted filesystems.

The common options are:

Option	Description
`-h`	Display sizes in a human-readable format (e.g., KB, MB, GB).
`-T`	Display the file system type along with other information.

The following image shows an example of the **df** command output in a human-readable format:



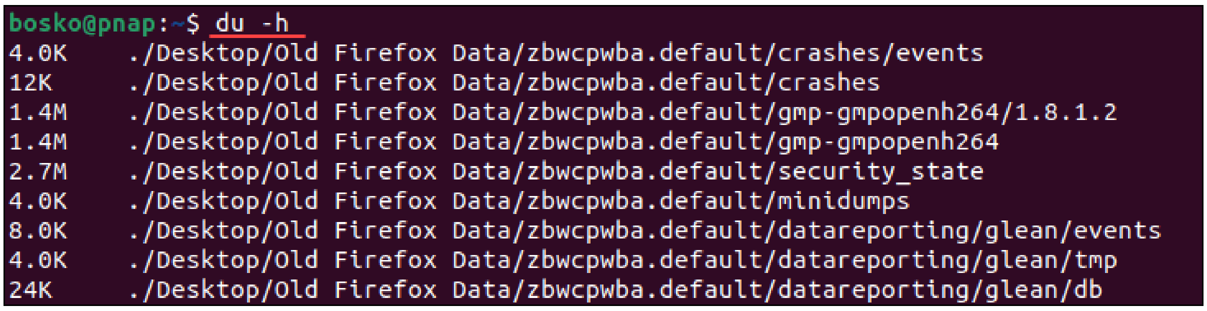
```
bosko@pnap:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           392M  1.7M  390M   1% /run
/dev/sda3        49G   20G   27G  43% /
tmpfs           2.0G     0  2.0G   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
/dev/sda2       512M  6.1M  506M   2% /boot/efi
tmpfs           392M  112K  391M   1% /run/user/1000
bosko@pnap:~$
```

du Command

The du command estimates and displays the disk space usage of files and directories. By default, **du** provides the space used by the specified files and directories in bytes. It is useful for identifying which files or directories

`-h`	Display sizes in a human-readable format (e.g., KB, MB, GB).
`-s`	Display only the total size for each argument (directory or file).

The following image is a partial **du** command output:



```
bosko@pnap:~$ du -h
4.0K    ./Desktop/Old Firefox Data/zbwcpwba.default/crashes/events
12K     ./Desktop/Old Firefox Data/zbwcpwba.default/crashes
1.4M    ./Desktop/Old Firefox Data/zbwcpwba.default/gmp-gmpopenh264/1.8.1.2
1.4M    ./Desktop/Old Firefox Data/zbwcpwba.default/gmp-gmpopenh264
2.7M    ./Desktop/Old Firefox Data/zbwcpwba.default/security_state
4.0K    ./Desktop/Old Firefox Data/zbwcpwba.default/minidumps
8.0K    ./Desktop/Old Firefox Data/zbwcpwba.default/datareporting/glean/events
4.0K    ./Desktop/Old Firefox Data/zbwcpwba.default/datareporting/glean/tmp
24K     ./Desktop/Old Firefox Data/zbwcpwba.default/datareporting/glean/db
```

ps Command

The ps command lists processes currently running on the system. By default, it displays a list of processes associated with the current terminal session, including their process IDs (PIDs), resource utilization, and other attributes. Use the tool for monitoring system activity, identifying misbehaving processes, investigating resource consumption, and managing process-related tasks.

The command syntax is:

```
ps [options]
```

The `[options]` allow users to customize the output to show detailed information about all processes on the system, filter processes based on user or attributes, and format the output.

The most common options are:

Option	Description
`-e`	Display information about all processes on the system, regardless of whether they are associated with the current terminal session.

```
   8600 pts/2    00:00:00 bash
  38443 pts/2    00:00:00 ps
bosko@pnap:~$
```

Control Operations

Certain Bash commands are crucial for controlling processes and managing sessions effectively. Those commands allow users to manage running processes, review command history, and conclude sessions with precision, contributing to efficient and organized Linux usage. This section lists the key Bash commands for system management.

kill Command

The kill command is a Bash utility used to terminate processes by sending signals to them. It is useful for stopping misbehaving or unresponsive programs that might otherwise require restarting the system.

By specifying a Process ID (PID) or job control identifier along with a signal number or name, users can halt processes or prompt them to perform specific actions.

The command syntax is:

```
kill [options] pid…
```

`pid…` is the Process IDs (PIDs) of the processes you want to terminate. The command accepts one or more PIDs separated by spaces.

The commonly used options are:

Option	Description
`-l`	Lists available signals and their corresponding numeric values.
`-9`	Sends the **`SIGKILL`** signal to the target process.

The following image shows all acceptable **`kill`** command signals:

Important: Be careful when terminating system processes, as the action can cause system instability.

history Command

The history command is a Bash utility that displays a list of recently executed commands within a terminal session. It provides a chronological record of command-line inputs, along with their corresponding line numbers.

The command is useful for recalling and re-executing previously used commands, streamlining repetitive tasks, and maintaining a record of your actions for reference or documentation.

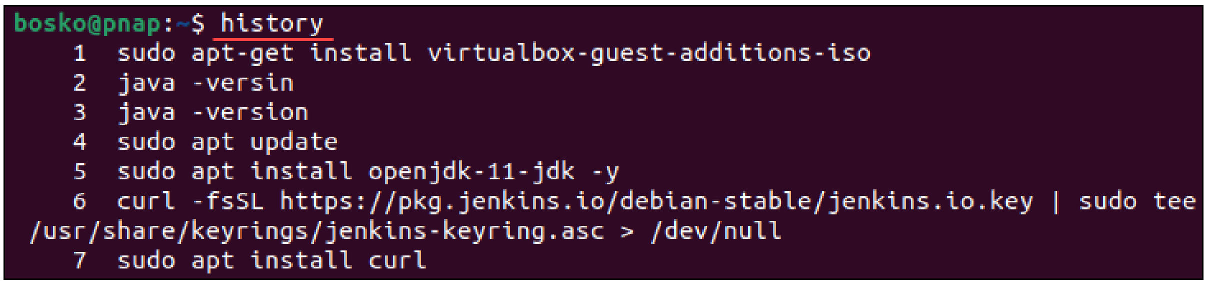
The command syntax is:

```
history [options]
```

The commonly used options are:

Option	Description
`-c`	Clears the command history, deleting all previously executed commands from the history list.
`-w`	Writes the current history list to the history file (*~/.bash_history* by default) and saves any changes made during the session.

The following example shows a partial **history** command output:



```
bosko@pnap:~$ history
    1  sudo apt-get install virtualbox-guest-additions-iso
    2  java -versin
    3  java -version
    4  sudo apt update
    5  sudo apt install openjdk-11-jdk -y
    6  curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo tee
 /usr/share/keyrings/jenkins-keyring.asc > /dev/null
    7  sudo apt install curl
```

exit Command

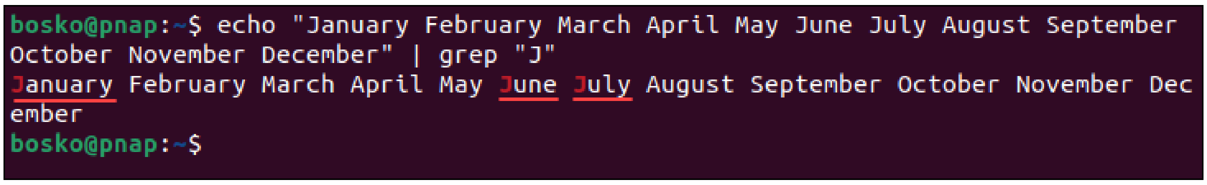
Pipe Utility

The **|** (pipe) utility is a powerful tool that allows users to connect the standard output of one command to the standard input of another, thus enabling the flow of data between commands. It is beneficial when performing tasks that involve multiple commands, data transformation, filtering, or data aggregation.

The basic syntax is:

```
[command1] | [command 2]
```

In the following example, the output of the echo command is piped into **`grep`**, which filters out the months that start with "J":



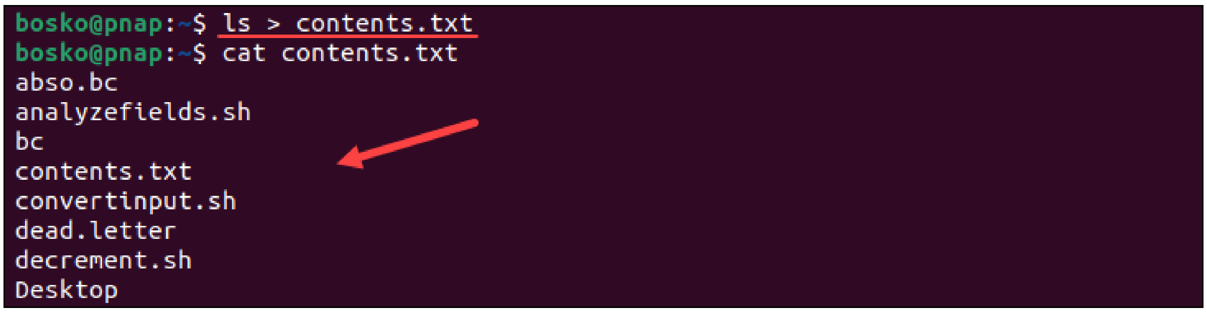
```
bosko@pnap:~$ echo "January February March April May June July August September October November December" | grep "J"
January February March April May June July August September October November Dec
ember
bosko@pnap:~$
```

Redirect Operator

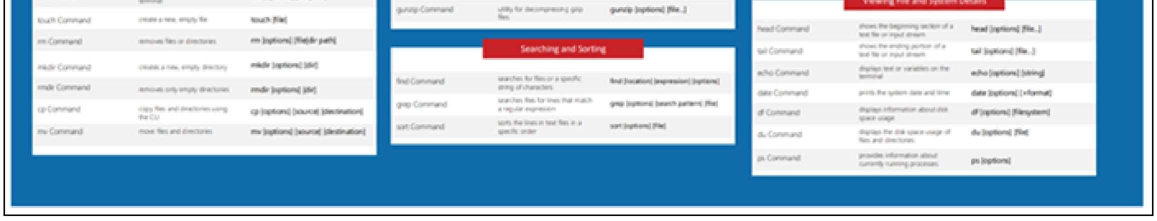
The redirect operator (**>**, **<**, **>>**, **2>**, etc.) is a mechanism that allows users to control the input and output of commands. **>** redirects the standard output of a command to a file, overwriting its content if the file already exists. The **>>** operator appends the standard output to the specified file.

< operator redirects the standard input of a command from a file. The **2>** operator redirects standard error messages to a file. These operators enable users to manage data flow, perform file-based operations, and ensure precise error handling.

In the following example, the output of the **`ls`** command is redirected to a new file called *contents.txt*. After redirection, check the file's contents using **`cat`**:



```
bosko@pnap:~$ ls > contents.txt
bosko@pnap:~$ cat contents.txt
abso.bc
analyzefields.sh
bc
contents.txt
convertinput.sh
dead.letter
decrement.sh
Desktop
```



touch Command	create a new, empty file	touch [file]
rm Command	removes files or directories	rm [options] [file/dir path]
mkdir Command	create a new, empty directory	mkdir [options] [dir]
rmdir Command	removes only empty directories	rmdir [options] [dir]
cp Command	copy files and directories using the CLI	cp [options] [source] [destination]
mv Command	move files and directories	mv [options] [source] [destination]

gunzip Command	utility for decompressing gzip files	gunzip [options] [file...]

Searching and Sorting

find Command	searches for files or a specific string of characters	find [location] [expression] [options]
grep Command	searches files for lines that match a regular expression	grep [options] [search pattern] [file]
sort Command	sorts the lines in text files in a specific order	sort [options] [file]

head Command	shows the beginning section of a text file or input stream	head [options] [file...]
tail Command	shows the ending portion of a text file or input stream	tail [options] [file...]
echo Command	displays text or variables on the terminal	echo [options] [string]
date Command	prints the system date and time	date [options] [+format]
df Command	displays information about disk space usage	df [options] [filesystem]
du Command	displays the disk space usage of files and directories	du [options] [file]
ps Command	provides information about currently running processes	ps [options]

Conclusion

This article has listed the 30 most common Bash commands any Linux user should know. Use the commands to improve your file system management, automate and facilitate scripting, and level up your file management skills. We also provided the Bash commands PDF cheat sheet for your future reference.

For more useful commands, check out our ultimate list of Linux commands all users should know.

Was this article helpful? Yes No

Bosko Marijan

Having worked as an educator and content writer, combined with his lifelong passion for all things high-tech, Bosko strives to simplify intricate concepts and make them user-friendly. That has led him to technical writing at PhoenixNAP, where he continues his mission of spreading knowledge.

Next you should read

SysAdmin, Web Servers

Linux Commands Cheat Sheet: With Examples

November 2, 2023

PATH is an environmental variable that shows Linux where to search for executables. Read this guide and learn how to add a directory to PATH.

READ MORE

DevOps and Development, SysAdmin

Git Commands Cheat Sheet

March 10, 2020

Git has a plethora of commands for managing project history. This article lists out all the important commands every developer will need.

READ MORE

SysAdmin

Mac Terminal Commands

May 18, 2023

This article presents a comprehensive list of Mac terminal