



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/)

Visit <http://learnbyexample.net/> for more information

Linux: Introduction

A brief idea as to what Linux is and how to get started with using the command line.

What is Linux?

A free and open source operating system. Free as in free air. Open source meaning one can download the source code and modify as needed. Which has resulted in [hundreds of known distributions](#). To give an analogy, take cars – low priced Nano to high end Mercedes exist to serve different needs of end users.

Linux Kernel (used in Android OS) and Linux Server (more widely used than Windows Server) are prevalent than the Linux operating system. The material here gets you started with using the OS on your PC or laptop – commonly referred as Distros.

▶ Strictly speaking, Linux is a Kernel and [GNU/Linux](#) is operating system

Why Linux?

- ➔ Speed, Security, Stability
- ➔ Highly configurable
- ➔ Well defined hierarchy and permissions to allow networking across different groups and sites
- ➔ Strong set of commands and user configurability to automate repetitive manual tasks

Where is Linux deployed?

- Servers (also, 95% of top 500 Supercomputers)
- Embedded Systems
 - ➔ Mobile phones, Tablets, TV, etc
 - ➔ Android – built on top of Linux kernel
 - ➔ Google Chrome OS
 - ➔ iOS – Unix based
- Personal and Enterprise Computers
 - ➔ [Ubuntu](#), [Linux Mint](#), [Debian](#), [Fedora](#), [Red Hat Enterprise Linux](#), [openSUSE](#)
- And many more uses – all thanks to being open source

Lightweight Linux Distros

What is the download size of popular web browsers like Firefox and Chrome? 40+ MB.

What is the size of popular light weight Linux Distros? ~15 MB

[Arch Linux](#) and [Tiny Core Linux](#) are amongst the most widely used for building light weight Linux variants to one's requirements.

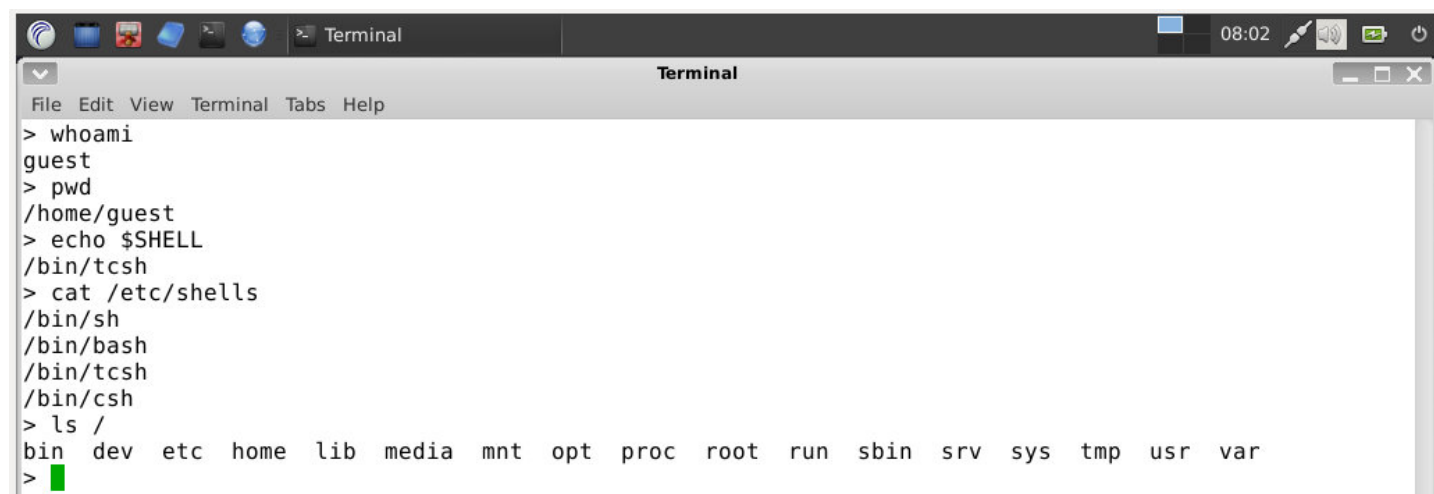
Download and installation instructions for a lightweight Porteus distro [here](#)

Suggested Reading

- [Wiki entry for Linux](#)
- [Usage Share of Operating Systems](#)
- [Popular Linux Distros](#)
- [Statistics of various Linux Distros](#)
- [Light Weight Linux Distros](#) (Article comments are good too)

Linux Commands: Introduction

For any thing that is repetitive or programmable, there likely is a Linux command. If you do not know any related command, ask your peers or search online before you start writing a script. Just remember that Unix was first implemented way back in 1969, there are commands for just about anything.



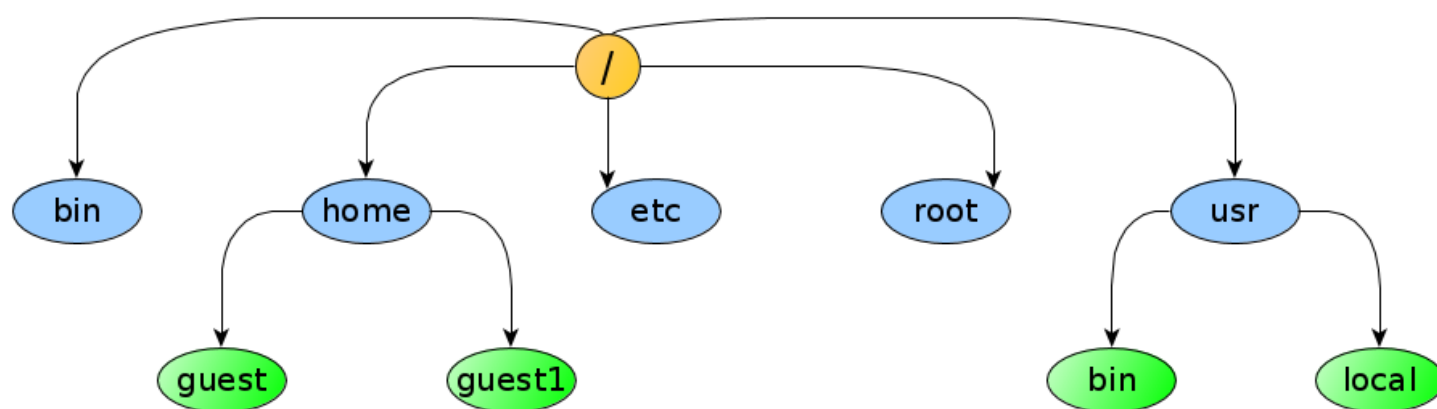
```
> whoami
guest
> pwd
/home/guest
> echo $SHELL
/bin/tcsh
> cat /etc/shells
/bin/sh
/bin/bash
/bin/tcsh
/bin/csh
> ls /
bin  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
>
```

Sample commands and their outputs

Biggest starting trouble with Linux (for those accustomed to GUI) is the sudden trouble of interacting with the computer using text commands. Once the user gets comfortable in a week or two, things seem very systematic and GUI becomes ill suited for frequent tasks. With continuous use, we tend to remember various commands better. Short-cuts (aliases, auto-completion, etc) reduce typing massively.

- ▶ Commands presented here are GNU specific and generally behave similarly across distros
- ▶ Commands in GNU/Linux are usually different (and mostly easier to use) than [POSIX](#) (**P**ortable **O**perating **S**ystem **I**nterface) standards set for Unix like systems
- ▶ If any command is not found in a particular distro, either it has to be manually installed or not available for that distro
- ▶ Some commands may behave differently across distro or even across different shells (ex: **which** command in tcsh & bash). **man** command or online search will help in such cases

File System



Sample Linux File System

Before we dive into ocean of Linux commands, let's get a brief how Linux directories are organized. If you've used Windows, you would be familiar with 'C:' 'D:' etc. In Linux, directory structure starts with / symbol, which is referred as the **root** directory. Use **man hier** command to get an overview. Also, check out these:

- [video tutorial](#)
- [overview of file system](#)

Command Line Interface - CLI

Shell and Terminal are sometimes interchangeably used to mean the same thing – a prompt where user enters and execute commands. They are [quite different](#) and in simple terms:

- ▶ Shell is a program (command line interpreter) that facilitates interaction between user and Kernel
 - ▶ Terminal facilitates user interaction with Shell and displays output
- Type **cat /etc/shells** and press Enter to know which shells are available
 - **echo \$SHELL** to know the current shell

The default interactive shell on most Linux distros is the **bash** shell, and used in this material unless otherwise specified

Command Help

- **man command** → provides help on syntax, options, examples, exit status etc (ex: **man ls**)
 - press **q** key to quit the man page and **h** key to get help
 - for GNU/Linux commands, complete manual is usually available with **info command**
 - use **help command** for built-in commands like **cd**, **type**, **compgen**, etc
- **whereis command** → gives where the binary and man page of command are located (ex: **whereis man**)
- **whatis command** → single line description of command (ex: **whatis bash**)
- **type command** → whether the command is shell built-in, alias or path of command (ex: **type grep**)
 - **which command** is better suited for **tcsh** shell
- **history** → recently used commands
- Lots of free help → Online search, linux communities, ebooks, colleagues!

Command aliases

- **alias** → display all aliases declared for the current shell
- **alias grep='grep --color=auto'** → **grep** command aliased as **grep --color=auto**
 - **alias grep 'grep --color=auto'** → **tcsh** syntax
- ▶ **\grep** → override alias declaration and execute original command

Utility Commands

- **date** → display current time with date
- **cal** → display current month calendar
- **expr 1 + 2** → evaluate expressions, special shell characters should be escaped with \ (ex: **expr 2 * 3**)
- **bc** → calculator, type **bc** and enter, evaluate any number of expressions line by line, type **quit** to exit
- **seq** → print sequence of numbers, by default in ascending order with increments of 1 (ex: **seq 10**, **seq 10 -1 5**, **seq 2 2 100**)

Suggested Reading

- [Linux general purpose utility commands](#)
- [Beginners guide to Linux Commands](#)
- [which, whatis, whereis examples](#)
- [Difference between terminal, shell, tty, console](#)

Linux Commands: Handling Files and Folders

Let's look at commonly used commands to move around directories, create and modify files and folders. For certain commands, a list of commonly used options are also given. Always use man command (example: **man ls**) to read about a new command and get to know its options. Or search online for best uses/options of the particular command

whoami

Prints username

▶ **who** command displays list of all users currently logged on the machine

pwd

Absolute path of present working directory. Used to know which is the current directory and often to copy the path to be pasted in a script

▶ Path is absolute if it starts with / and relative if not

clear

Clears the terminal screen

▶ Pressing **Ctrl+I** will retain already typed command and clear the remaining screen

ls

List directory contents of the path specified. When no option is given, lists contents of current directory

Options

- **-a** list hidden files also
- **-l** list in single column (number one, not lowercase of letter L)
- **-l** list contents with extra details about the files (lowercase of letter L, not number one)
- **-h** display file sizes in human readable format
- **-t** sort based on time
- **-r** reverse sorting order
- **-S** sort by file size (directory is treated as file and doesn't display size used by directory)
- **-F** append indicator like / for directories, * for executable, etc
- **--color=auto** list contents with different color for directories, executables, etc
- **-R** recursively display sub-directories

▶ Options can be combined

Examples

- **ls** → list contents of current directory
- **ls /home** → list contents of directory home present under the root directory (absolute path specified)
- **ls ../** → list contents of directory one hierarchy above (relative path specified)
- **ls -ltr** → list files of current directory with details sorted such that latest file is displayed last

cd

Change directory, i.e go to a specified path

▶ cd is a shell built-in command, so use **help cd** instead of **man**

Examples

- **cd /etc** → go to 'etc' directory under root folder (absolute path specified)
- **cd -** → switch back to previous working directory
- **cd ~/** or **cd ~** or **cd** → go to home directory (as specified by HOME environment variable)
- **cd ..** → go one hierarchy back (relative path specified)
- **cd ../../** → two hierarchy back (relative path specified)

mkdir

Create directory in the specified path

▶ **touch filename** creates an empty file at the specified path (it is a hack, **touch** has other functionalities)

Examples

- **mkdir project_adder** → create folder project_adder in current directory
- **mkdir project_adder/report** → create folder report in project_adder directory
- **mkdir -p project_adder/report** → create both project_adder and report directories in one shot (if project_adder already exists, it won't be affected)
- **mkdir /home/guest1** → add a home directory for user guest1

rm

Remove files and directories

Options

- **-r** remove recursively, used for removing directories
- **-f** force remove without prompt for non-existing files and write protected files (provided immediate parent directory has write permission)
- **-i** prompt before every removal

Examples

- **rm project_adder/power.log** → remove file power.log from project_adder directory
- **rm -r project_adder** → remove folder project_adder from current directory even if non-empty
- ▶ **rmdir project_tmp** → remove project_tmp folder provided it is empty (same as **rm -d**)
- ▶ Ubuntu users can use **gvfs-trash** command to send items to trash instead of permanent deletion
- ▶ Files removed using **rm** can still be recovered with time/skill. Use **shred** command to overwrite files

cp

Copy files and directories from one path to another. The destination path is always specified as the last argument. More than one source file/folder can be specified if destination is a directory

▶ **rsync** is a remote sync command, which has capabilities like merging changes to destination (instead of copying entire file(s)) - [examples](#)

Options

- **-r** copy recursively, used for copying directories
- **-i** prompt before overwriting

Examples

- **cp** /home/raja/Raja_resume.doc Ravi_resume.doc → create a copy of file Raja_resume.doc as Ravi_resume.doc in your current directory
- **cp -r** /home/guest1/proj_matlab ~/proj_matlab_bug_test → copy proj_matlab to your home directory
- **cp** report/output.log report/timing.log . → copy files output.log and timing.log to current directory (single dot represents current directory and double dot represents one hierarchy above)

mv

Move or rename files and directories. The destination path is always specified as the last argument. More than one source file/folder can be specified if destination is a directory

Options

- **-f** don't prompt for overwriting and moving write protected files (provided immediate parent directory has write permission)
- **-i** prompt before overwriting

Examples

- **mv** project_adder project_lowpower_adder → rename file or folder
- **mv** power.log timing.log area.log project_multiplier/result → move the specified files to folder result

rename

Rename multiple files using Perl regular expression

Options

- **-f** overwrite existing files
- **-n** show matching input files without actually renaming them

Examples

- **rename** 's/txt\$/log/' *txt → rename all file in current directory ending with 'txt' to 'log'

ln

Create hard or soft link of file or folder. Soft link is similar to short-cuts created in Windows. Hard link is like same file with different name, same timestamp and permissions of original file. Hard links can be moved to another directory after creation, will still have content even when original file is deleted. On the other hand, soft links have their own timestamps and permissions, it cannot be moved to another folder unless the link creation was done using full path and of course becomes a dead link when original file is deleted. More differences [here](#)

Examples

- **ln -s** results/report.log report.log.softlink → create a symbolic link of report.log from results folder to current directory
- **ln** results/report.log report.log.hardlink → create a hard link of report.log from results folder to current directory, will not lose content even if results/report.log file is deleted

► **unlink** or **rm** commands can be used to delete links

tar and gzip

tar is archiving utility. The archived file is same size as combined sizes of archived files. Usually so often combined with compression utility like gzip that there is a way to do it just using tar command.

Examples

- **tar -cvf backup_mar15.tar project results** → create backup_mar15.tar of files/folders project and results. -v option stands for verbose, i.e displays all the files and directories being archived
- **gzip backup_mar15.tar** → overwrites backup_mar15.tar with backup_mar15.tar.gz, a compressed version
- **tar cvfz backup_mar15.tar.gz project results** → create backup_mar15.tar and overwrite with backup_mar15.tar.gz, no need of separate gzip command (Note that - is not used)
- **gunzip backup_mar15.tar.gz** → uncompress and overwrite as backup_mar15.tar
- **tar -xvf backup_mar15.tar** → extract archived files to current directory
- **tar xvfz backup_mar15.tar.gz** → uncompress and extract archived files to current directory (Note that - is not used)
- **zcat story.txt.gz** → uncompress and display file contents on standard output. There are other commands as well like **zgrep**, **zdiff**, etc to work on compressed files

Linux Commands: Working with Files and Folders

In this we see how to display contents of a file, search within files, search for files, what are the permissions for files and directories and how to change them to our requirements

cat

what is cat - Concatenate files and print on the standard output. Standard output in our context is the Terminal screen. Often, the output is redirected to new file using redirection symbol `>` or given as input to another command using pipe symbol `|`

- ▶ cat command is NOT suitable for displaying large files on the Terminal
- ▶ use **tac** command to display the content in reversed order i.e last line is shown first and first line as last

Options

- **-n** number output lines
- **-s** squeeze repeated empty lines into single empty line

Examples

- **cat > sample.txt** → create a new file for writing, use Ctrl+C on a newline to save and quit
- **cat sample.txt** → display the contents of file sample.txt
- **cat power.log timing.log area.log > report.log** → concatenate the contents of all three files and save to new file report.log (if report.log already exists, it will be overwritten)
- **cat power.log timing.log area.log | wc -l** → print total number of lines of the three files

```
> cat > sample.txt
This is an example of adding text to a new file using cat command.
After typing the text, one has to press Ctrl+C in a newline to save and quit.
^C
> cat sample.txt
This is an example of adding text to a new file using cat command.
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> ls -lh sample.txt
-rw-r--r-- 1 guest guest 145 Mar 12 12:25 sample.txt
> wc sample.txt
  2  30 145 sample.txt
> █
```

Using cat command

less

Display contents of a file, automatically fits to size of Terminal, allows scrolling in either direction and other options for effective viewing. Usually, man command uses less command to display the help page. The navigation options are similar to **vi** editor

- ▶ less is an [advanced version](#) of **more** command

Navigating Options

- **g** go to start of file
- **G** go to end of file
- **q** quit
- **/pattern** search for the given pattern
- **n** go to next pattern in forward direction
- **N** go to next pattern in backward direction

- **h** help

Example

- **less large_filename** → display contents of file large_filename using less command

tail

Used to display last portions of a file. By default, displays last 10 lines of a file.

Examples

- **tail report.log** → display last 10 lines of report.log
- **tail -20 report.log** → display last 20 lines of report.log
- **tail -5 report.log** → display last 5 lines of report.log
- **tail power.log timing.log** → display last 10 lines of both files preceded by filename header
- **tail -q power.log timing.log > result.log** → save last 10 lines of both files without filename to result.log (option -q stands for quiet)
- **tail -n +3 report.log** → display all lines starting from 3rd line (i.e all lines except first two lines)

head

This command is used to display starting portions of a file. By default, displays starting 10 lines of a file.

Examples

- **head report.log** → display first 10 lines of report.log
- **head -20 report.log | tail report.log** → display lines numbered 11-20 of report.log
- **tail -20 report.log | head report.log** → display last but 10 lines of report.log
- ▶ **head -n -2 report.log** → display all but last 2 lines of report.log (using negative number -2, may not work in some distros)

grep

Stands for **Global Regular Expressions Print**. Search for a pattern in given file(s). Often used to know whether a particular word or pattern is present (or not present) in files of interest, name of files containing the pattern, etc. By default, matching is performed any part of a line, options are used to add restrictions

Options

- **--color=auto** display the matched pattern, file name, etc with color
- **-i** ignore case while matching
- **-v** matches everything other than pattern
- **-r** recursively search all files in specified folder(s)
- **-n** print also line number(s) of matched pattern
- **-c** count of number of 'lines' having the match
- **-l** print only the filename(s) with matching pattern
- **-L** print filename(s) NOT having the pattern
- **-w** match exact word
- **-o** print only matching parts
- **-A number** print matching line and 'number' of lines after the matched line
- **-B number** print matching line and 'number' of lines before the matched line
- **-C number** print matching line and 'number' of lines before & after the matched line

- **-m number** upper limit of matched lines specified by 'number'

Examples

- **grep area report.log** → will print all line containing the word area in report.log
- **grep 'adder power' report.log** → will print lines containing adder power (quotes are used to specify multiple words)
- **man grep | grep -i 'EXIT STATUS' -A 5** → will print matched line and 5 lines after containing the words 'exit status' independent of case (man pages of commands like expr do not have EXIT STATUS as heading like grep does)
- **grep -m 5 -i error report.log** → will print maximum of 5 lines containing the word error (ignoring case)
- **grep "\$HOME" /etc/passwd** → will print lines matching the value of variable \$HOME
- ▶ To group multiple words, use single quotation '
- ▶ To use substitution, like HOME environment variable, use double quotation ''

```

Terminal
File Edit View Terminal Tabs Help
> cat sample.txt
This is an example of adding text to a new file using cat command.
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> alias grep 'grep --color'
> grep add sample.txt
This is an example of adding text to a new file using cat command.
> grep 'has to' sample.txt
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> grep -v add sample.txt
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> grep text sample.txt
This is an example of adding text to a new file using cat command.
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> grep -w an sample.txt
This is an example of adding text to a new file using cat command.
> grep an sample.txt
This is an example of adding text to a new file using cat command.
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> grep -i t sample.txt
This is an example of adding text to a new file using cat command.
After typing the text, one has to press Ctrl+C in a newline to save and quit.
> grep -i '^t' sample.txt
This is an example of adding text to a new file using cat command.
> grep -c text sample.txt
2
> grep 'command.$' sample.txt
This is an example of adding text to a new file using cat command.
> grep "command.$" sample.txt
Illegal variable name.
>

```

grep command examples

There was something called 'Regular Expressions' in the expansion of grep. What does that mean? Suppose you have a file with some lines containing alphabets and others containing numbers. How would you extract the lines with alphabets or those with numbers? Regular Expressions to rescue

- ▶ **egrep** command is same as **grep -E** and **fgrep** is same as **grep -F** (Check out this excellent [answer](#))

Regular Expression Examples

- **grep -i '[a-z]' report.log** → will print all lines having atleast one alphabet
- **grep '[0-9]' report.log** → will print all lines having atleast one number
- **grep -P '\d' report.log** → will print all lines having atleast one number
- **grep '\w' report.log** → will print all lines having alphabets of either case, numbers or the underscore
- **grep 'area|power' report.log** → will match lines containing either area or power or both

- **grep** -E 'area|power' report.log → -E option, | is treated as meta-character with special meaning
- **grep** -E 'hand(y|ful)' short_story.txt → match either handy or handful or both

Regular Expression Options

- **[]** match any of the character specified within [], use **[^]** to invert the selection
- **\w** match alphabets (both upper & lower cases), numbers and **_** i.e short cut for **[a-zA-Z0-9_]**
- **\W** opposite of **\w** i.e short cut for **[^a-zA-Z0-9_]**
- **|** matches either of the given patterns
- **()** patterns within () are grouped and treated as one pattern
- **^** match from start of line
- **\$** match end of line
- **-P** treat the regex as Perl regular expression

Suggested Reading

- [15 practical grep examples](#)
- [Difference between grep, egrep and fgrep](#)
- [grep tutorial](#)

find

Search for files by variety of criteria like name, empty files, type of file like regular or hidden files/folders, size of files, permissions of files, etc. It is one of the most complicated command whose syntax might be difficult to get used to

Examples

- **find . -iname 'power.log'** → search and print path of file named power.log (ignoring case) in current directory and its sub-directories (the path argument (.) in this example) is optional when searching the current directory)
 - **find . | grep -i 'power.log'** → easier to remember alternative
- **find -name '*.log'** → search and print path of all files whose name ends with log in current directory
- **find -not -name '*.log'** → print path of all files whose name does NOT end with log in current directory
- **find /home/guest1/proj -type f** → print path of all regular files found in specified directory
- **find /home/guest1/proj -type d** → print path of all directories found in specified directory
- **find /home/guest1/proj -type f -name '.*'** → print path of all hidden files
- **find report -name '*.log' -exec rm {} \;** → delete all files ending with .log in report and its sub-folders
 - **find report -name '*.log' | xargs rm** → alternative easier to remember syntax (but slower)
- **find -name '*.txt' -exec wc {} +** → list of files ending with txt are all passed as argument to single wc command instead of executing wc command for every file

Suggested Reading

- [Examples of using find command](#)
- [Collection of find examples](#)

locate

Faster alternative to find command when searching for a file by its name. It is based on a database, updated by a cron job. So, newer files may not be displayed. Use this command if it is available and you remember some part of filename created a day ago or older. Very useful if one has to search entire filesystem in which case find command might take a very very long time compared to second or less for locate

Examples

- **locate power.log** → print path of files whose name contain power.log in the whole Linux filesystem
- **locate -b 'power.log'** → print path of files matching the name power.log exactly

WC

Display number of lines, words and characters of given file(s)

Examples

- **wc power.log** → outputs no. of lines, words and characters separated by space and followed by filename
- **wc -l power.log** → outputs no. of lines followed by filename
- **wc -w power.log** → outputs no. of words followed by filename
- **wc -c power.log** → outputs no. of characters followed by filename
- **wc -l < power.log** → output only the number of lines

du

Display size of specified file(s) and folder(s)

▶ du command is useful for small folders, not for large ones or file systems.

Examples

- **du project_report** → display size (default unit is 1024 bytes) of folder project_report as well as it sub-directories
- **du --si project_report** → display size (unit is 1000 bytes) of folder project_report as well as it sub-directories
- **du -h project_report** → display size in human readable format
- **du -sh project_report** → display size only for project_report folder in human readable format
- **du -sm * | sort -n** → sort files and folders of current directory, numbers displayed are in Megabytes (use **sort -nr** to reverse sort order, i.e largest at top)
- **du -sh * | sort -h** → sort files and folders of current directory, output displayed in human-readable format

df

Display size of file systems, how much is used and how much is available

Examples

- **df -h** → display usage statistics of all available file systems
- **df -h .** → display usage of only the file system where the current working directory is located

chmod

One of the most important Linux command in terms of security and allocating appropriate permissions depending on type of file/folder and their functionalities. When you use **ls -l** command, the first 10 characters displayed are related to type of file and its permissions.

```

guest@porteus:~/shell_scripting$ ls
total 4.0K
-rwxr-xr-x 1 guest guest 107 Apr 10 08:06 hello_user.sh*
-rwxr-xr-x 1 guest guest  0 Apr 10 08:22 for_loop.sh*
guest@porteus:~/shell_scripting$ chmod -x hello_user.sh
guest@porteus:~/shell_scripting$ ls
total 4.0K
-rw-r--r-- 1 guest guest 107 Apr 10 08:06 hello_user.sh
-rwxr-xr-x 1 guest guest  0 Apr 10 08:22 for_loop.sh*
guest@porteus:~/shell_scripting$ chmod 755 hello_user.sh
guest@porteus:~/shell_scripting$ ls
total 4.0K
-rwxr-xr-x 1 guest guest 107 Apr 10 08:06 hello_user.sh*
-rwxr-xr-x 1 guest guest  0 Apr 10 08:22 for_loop.sh*
guest@porteus:~/shell_scripting$

```

Linux File Permission

File Type

First character indicates the file type. Three common file types are

- - regular file
- d directory
- l linked file

File Permissions

The other 9 characters represent three sets of file permissions for user, group and others – in that order

- **user** → who is the owner of file (**u**)
- **group** → which is the group owner (**g**)
- **others** → everyone else (**o**)

Permission characters and values

Character	Meaning	Value	File	Folder
r	read	4	Can be read	Can read folder contents
w	write	2	Can be modified	Can add/remove folder contents
x	execute	1	Can be executed (ex: binary files, scripts)	Can list info about folder contents
-	disabled	0	Used to indicate which permissions are disabled	

Permissions for regular files are straight forward to understand. For directories, remember that 'w' gives permission to add/remove files, while 'r' and 'x' are almost always used together

Examples

- **chmod 766 script.sh** → rwx for user, rw for group and others
- **chmod 755 proj_folder** → rwx for user, rx for group and others
- **chmod +w script.sh** → add w for user and group, doesn't affect r and x permissions
- **chmod -x report.log** → remove x for user, group and others
- **chmod +w -R proj_folder** → add w for proj_folder as well all of its sub-directories and files
- **chmod o-r script.sh** → remove r only for others
- **chmod ug+x script.sh** → add x for user and group

- ▶ **+r, -r, +x, -x** without u/g/o qualifier affects all the three categories
- ▶ **+w, -w** without u/g/o qualifier affects only user and group categories
- ▶ **chmod -w -R proj_folder** → to make a directory write protected, always remember to use -R option, otherwise the contents of sub-directories can still be modified i.e the folder properties affect only the immediate files/directories

Suggested Reading

- [Linux File Permissions](#)

xargs

Used usually for passing output of a command as input arguments to another command. Good examples are given in **man xargs**

Examples

- **find | grep '.log\$' | xargs rm** → remove all files ending with '.log' from current directory and sub-directories
- **cat dir_list.txt | xargs mkdir** → create directories as listed in dir_list.txt file
- **ls -t *txt | head -1 | xargs gvim** → open the latest modified txt file in current directory

touch

Used to change file time stamps. But if file doesn't exist, the command will create an empty file with the name provided. Both features are quite useful. Some program may require a particular file to be present to work, empty file might even be a valid argument. In such cases, a pre-processing script can use touch command to scan the destination directories and create empty file if needed. Similarly, some programs may behave differently according to the time stamps of two or more files – while debugging in such an environment, user might want to just change the time stamp of files.

Examples

- **touch new_file.txt** → create an empty file if it doesn't exist in current directory
- **touch report.log** → change the time stamp of report.log to current time (assuming report.log already exists in current directory)

file

Returns the type of file. Linux doesn't require filename extensions like Windows does. This command comes in handy to identify the file type

identify

Although file command can also give information like pixel dimensions and image type, identify is more reliable command for images and gives complete format information

Linux Commands: Text Processing

There are commands for almost anything that need to be done in Linux, but the rich set of text processing commands is simply comprehensive and time saving. Knowing even their existence is enough to avoid the need of writing yet another script (which takes time and effort plus debugging) – a trap which many beginners fall into. An extensive list of commands and examples can be found [here](#)

sort

As the name implies, this command is used to sort files. How about alphabetic sort and numeric sort? Possible. How about sorting a particular column? Possible. Prioritized multiple sorting order? Possible. Randomize? Unique? Just about any sorting need is catered by this powerful command

Options

- **-R** random sort
- **-r** reverse the sort order
- **-o filename** redirect sorted result to specified filename, very useful to sort a file inplace
- **-n** sort numerically
- **-u** sort unique, i.e remove duplicates
- **-b** ignore leading white-spaces of a line while sorting

Examples

- **sort dir_list.txt** → display sorted file on standard output
- **sort -bn numbers.txt -o numbers.txt** → sort numbers.txt numerically and overwrite the file with sorted output
- **sort -R crypto_keys.txt -o crypto_keys_random.txt** → sort randomly and write to new file
▶ **shuf crypto_keys.txt -o crypto_keys_random.txt** → equivalent way to do this

Suggested Reading

- [Sort like a master](#)
- [Sort a file inplace](#)

uniq

This command is more specific to recognizing duplicates. Usually requires a sorted input for most/all functionalities as the comparison is made on adjacent lines only

Options

- **-d** print only duplicate lines
- **-c** prefix count to occurrences
- **-u** print only unique lines

Examples

- **sort test_list.txt | uniq** → outputs lines of test_list.txt in sorted order with duplicate lines removed
 - **sort -u test_list.txt** → equivalent command
 - **uniq <(sort test_list.txt)** → using process substitution, not available in **tcsh**
- **uniq -d sorted_list.txt** → print only duplicate lines, use -c option to prefix number of times a line is repeated

- **uniq -u sorted_list.txt** → print only unique lines, repeated lines are ignored

comm

Compare two sorted files line by line. Without any options, it prints output in three columns – lines unique to file1, line unique to file2 and lines common to both files.txt

Options

- **-1** suppress lines unique to file1
- **-2** suppress lines unique to file2
- **-3** suppress lines common to both files

Examples

- **comm -23 sorted_file1.txt sorted_file2.txt** → print lines unique to sorted_file1.txt
- **comm -13 sorted_file1.txt sorted_file2.txt** → print lines unique to sorted_file2.txt
- **comm -12 sorted_file1.txt sorted_file2.txt** → print lines common to both files

Using Process Substitution

- **comm -23 <(sort file1.txt) <(sort file2.txt)** → [Bash Process Substitution](#) No need to sort the files first!
Note that there should be no space between < and (
- **bash -c 'comm -23 <(sort file1.txt) <(sort file2.txt)'** → process substitution feature is not yet available in tcsh shell, so use [bash -c](#) instead

cmp

Compares two files byte by byte. Very useful to compare binary files. If the two files are same, no output is displayed (exit status 0). If there is a difference, it prints the first difference – line number and byte location (exit status 1). Option **-s** allows to suppress the output – useful in scripts

diff

Compare two files line by line. Useful to compare text files. If the two files are same, no output is displayed (exit status 0). If there is a difference, it prints all the differences, which might not be desirable if files are too long

► For better results, sort the files to be compared, but may not be always desired

Options

- **-s** convey message when two files are same
- **-y** two column output
- **-i** ignore case while comparing
- **-w** ignore white-spaces
- **-r** recursively compare files between the two directories specified
- **-q** report if files differ, not the details of difference

Examples

- **diff -s test_list_mar2.txt test_list_mar3.txt** → compare two files
- **diff -s report.log bkp/mar10/** → no need to specify 2nd filename if names are same
- **diff -sr report/ bkp/mar10/report/** → recursively compare all matching filenames from given directories,

filenames not matching are also specified in output

- **diff -r report/ bkp/mar10/report/ | grep -w diff** → useful trick to get only names of mismatching files, and change the output line to use tkdiff (provided no mismatching lines contain the whole word diff)
▶ **diff -qr report/ bkp/mar10/report/** → should work too, see [this link](#) for detailed analysis and corner cases

Graphical diff

- **tkdiff** → visually show difference between two files ([About tkdiff](#), if you don't have tkdiff - [download here](#) extract and copy the tkdiff file to local bin folders like /usr/local/bin/)
- [other GUI diff tools](#)

pr

Convert text files for printing with header, footer, page numbers, etc. Double space a file or combine multiple files column wise. Good examples [here](#)

tr

Translate or delete characters

Options

- **-d** delete the specified character(s)
- **-c** complement set of character(s) to be replaced

Examples

- **tr a-z A-Z < test_list.txt** → convert lowercase to uppercase
- **tr -d ._ < test_list.txt** → delete the dot and underscore characters
- **tr a-z n-za-m < test_list.txt > encrypted_test_list.txt** → Encrypt by replacing every lowercase alphabet with 13th alphabet after it. Same command **tr a-z n-za-m** on encrypted text will decrypt it

sed and awk

sed and awk are two powerful commands, can be likened to programming languages. sed is shortform for Stream Editor, while awk derives its name from authors Alfred Aho, Peter Weinberger and Brian Kernighan. If a user is dealing with lots of text processing, it is prudent to learn sed and awk, or just awk (which can do almost everything sed can do). If lot of scripting is needed along with text processing, a full fledged scripting language like Perl or Python is recommended - but should still try to incorporate easy to use sed/awk constructs to save time and debug effort. Let's see some magic one liners with sed and awk

sed range definition

By default, sed acts on entire input contents. This can be refined to specific line number or a range defined by line numbers, search pattern or mix of the two. See [sed Regular Expression syntax](#) for reference

- **n,m** → range between nth line to mth line, including n and m. Use just n to specify only nth line. Use \$ to specify last line
- **/pattern/** → line matching pattern
- **n,/pattern/** → nth line to line matching pattern
- **/pattern/,m** → line matching pattern to mth line
- **/pattern1/,/pattern2/** → line matching pattern1 to line matching pattern2
- **/pattern/I** → line matching pattern – case insensitive

Examples

Search and Replace

- **sed -i 's/cat/dog/g' story.txt** → search and replace every occurrence of cat with dog in story.txt (-i option modifies the input file itself, i.e. in-place editing. g modifier is used to replace all occurrences in a line)
- **sed -i.bkp 's/cat/dog/g' story.txt** → in addition to in-place file editing, create backup file story.txt.bkp, so that if anything goes wrong, original file can be restored
- **sed -i '5,10s/cat/dog/g' story.txt** → search and replace every occurrence of cat (case insensitive due to modifier I) with dog in story.txt only in line numbers 5 to 10
- **sed '/cat/ s/animal/mammal/g' story.txt** → replace animal with mammal in all lines containing cat. Since -i option is not used, output is displayed on terminal and story.txt is not changed
- **sed -i -e 's/cat/dog/g' -e 's/lion/tiger/g' story.txt** → search and replace every occurrence of cat with dog and lion with tiger (any number of -e option can be used)
 - **sed -i 's/cat/dog/g ; s/lion/tiger/g' story.txt** → alternative syntax (spacing around ; is optional)
- **sed -r 's/(.*)/abc: \1 :xyz/' list.txt** → add prefix 'abc: ' and suffix ' :xyz' to every line of list.txt
- **sed -r -i "s/(.*)\$(pwd | xargs basename) \1/" *.txt** → use of double quotes allows to use output of other commands within \$. In this example, the current directory name is inserted at start of every line

Selective printing(p) or deletion(d)

- **sed '/cat/d' story.txt** → delete every line containing the text cat
- **sed '/cat/!d' story.txt** → delete every line NOT containing the text cat
- **sed '\$d' story.txt** → delete last line of the file
- **sed '2,5d' story.txt** → delete lines 2,3,4,5 of the file
- **sed '1,/test/d' dir_list.txt** → delete all lines from beginning of file to first occurrence of line containing test (the matched line is also deleted)
- **sed '/test,\$d' dir_list.txt** → delete all lines from line containing test to end of file
- **sed -n '5p' story.txt** → print 5th line (-n option overrides default print all lines behavior of sed)
 - use **sed -n '5q;d' story.txt** on large files. [Read more](#)
- **sed -n '/cat/p' story.txt** → print every line containing the text cat
 - equivalent to **sed '/cat/!d' story.txt**
- **sed -n '4,8!p' story.txt** → print all lines except lines 4 to 8
- **man grep | sed -n '/^ *exit status/I,/^\$/p'** → extract exit status information of a command from manual
- **man ls | sed -n '/^ *-F/,/^\$/p'** → extract information on command option from manual

awk one-liner general syntax

- **awk 'BEGIN {initialize} /pattern1/ {stmts} /pattern2/ {stmts}... END {finish}'**
 - **BEGIN {initialize}** → used to initialize variables (could be user defined or awk variables or both), executed once – optional block
 - **/pattern1/ {stmts} /pattern2/ {stmts}...** → action performed for every line of input, pattern is optional, more than one block {} can be used with/without pattern
 - **END {finish}** → perform action once at end of program – optional block
- [Reference sheet](#) for awk variables, statements, functions, etc

Examples for awk

- **awk 'BEGIN {FS = ":"} {print \$1}' /etc/passwd** → prints first column of /etc/passwd file. FS is awk's field separator variable. \$1, \$2, etc refer to 1st, 2nd field, etc \$0 holds entire line, \$NF holds last field
 - **awk -F":" '{print \$1}' /etc/passwd** → alternative syntax
- **awk '{print "abc: " \$0 " :xyz"}' list.txt** → add prefix 'abc: ' and suffix ' :xyz' to every line of list.txt
- **awk '{print FNR ": " \$0}' *.txt** → prefix line number and ': ' to all txt files in current directory
- **ls -l | awk 'NF > 2 {print \$1 " " \$NF}'** → print file permission and filename in two columns separated

by space. NF - awk variable for number of fields

- **ls -l | awk '/^d/ {print \$1 " " \$NF}'** → print file permission and filename only for directories
- **ls -l | grep '^-' | awk '{ total += \$5 } END { print total }' | numfmt --to=iec-i --suffix=B** → calculate the combined size of **regular files** in current directory and print in human readable format (powers of 1024). The first {} of awk works on all the input lines, total is a user defined variable name (any name can be given) whose initial value is 0 by default, \$5 is 5th field containing file size in Bytes. The {} following END keyword is executed only once after processing all the input lines. **Note** that FS hasn't been defined, which has default value of white-space

Suggested Reading

- [sed](#) and [awk](#) introduction
- [sed tutorial](#)
- [awk tutorial](#)
- [Easy to understand sed with examples](#) , [awk examples](#)
- [sed examples](#)
- [stackexchange sed examples](#)
- [more awk examples](#)
- [basic tutorial for grep, awk, sed](#)
- [sed](#) and [awk](#) books
- [sed](#) and [awk](#) reference sheets

cut

For columns operations with well defined delimiters, cut command can be used instead of sed/awk

Example

- **cut -d':' -f1 /etc/passwd** → prints first column of /etc/passwd file. -d option specifies delimiter character (in this case :) and -f option specifies which fields to print separated by commas (in this case field 1). Default delimiter is TAB character

paste

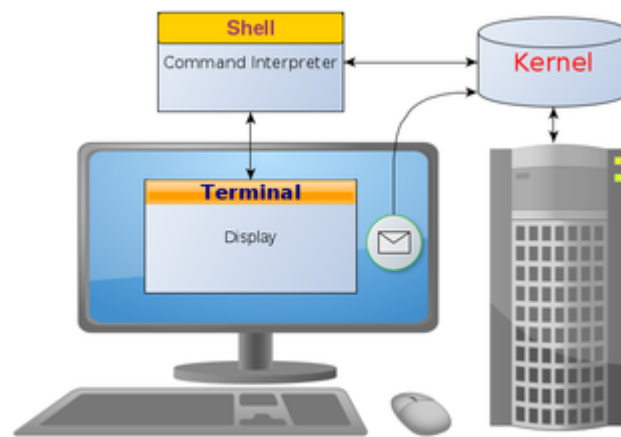
Combine two or more column oriented files into single file

Example

- **paste -d':' list1.txt list2.txt list3.txt > combined_list.txt** → combines the three list files into single file, the entries joined by : character. Default delimiter is TAB character

Linux: Shell

Shell basically facilitates the user to interact with Linux Kernel by working as '**command interpreter**'.



Shell and Kernel interaction

Like any indispensable software, Shell has undergone huge transformation from the days of basic 'sh' shell that was used in 1970s Unix. While bash is default shell in most distros and most commonly used, powerful and feature rich shells are still being developed and released. Here's a [comparison of command shells](#). bash shell has also been ported on Windows platform, for example [Cygwin](#) and [MinGW](#)

The material presented here is with focus on interactive shell, and not login shell.

What does Shell do?

- Interprets user commands
 - ➔ from terminal or from a file (shell scripts)
 - ➔ evaluate wildcards before passing command to Kernel
- Interacts with Kernel to execute the commands and convey outputs back to user
- Like a program, Shell has variables too which are used to customize aspects of its behavior

Popular shells

- sh → bourne shell (light weight Linux distros usually come with 'sh' shell only)
- bash → bourne again shell (one of the most popular shells)
- csh → C shell
- tcsh → tenex C shell
- ksh → Korn shell
- zsh → Z shell (bourne shell with improvements, including features from bash, tcsh, ksh)
- ▶ **cat /etc/shells** → displays list of login shells available in the current Linux distro
- ▶ **echo \$SHELL** → shows the path of login shell

Changing default shell

- **chsh -s /bin/tcsh** → the new shell has to be valid shell as stored in /etc/shells. For the change to become effective, the user has to log-off and then log-in
- Superuser (i.e root user) can change the entry corresponding to the username in **/etc/passwd** file

Suggested Reading

- [Shell, choosing shell and changing shells](#)
- [Features and differences between various shells](#)

- [Good article on command and syntax on different shells with examples](#)

Wildcards

It is easy to specify complete filenames as command arguments when they are few in number. But suppose, one has to delete 100s of log files, spread across different sub-directories, what to do? Wildcards to the rescue, provided the filenames have a commonality to exploit. We have already seen regular expressions used in commands like `grep`. Shell wildcards are very similar. Let's see some of the wildcard options available followed by examples

Options

- `*` → match any character, 0 or more times
- `?` → match any character exactly 1 time
- `[xyz]` → match a single character specified by characters enclosed in []
- `[a-z]` → match a single character specified by range of characters enclosed in []
- `[!aeiou]` → exclude a single character match of specified characters
- `[!0-9]` → exclude a single character match of specified range of characters
- `{word1,word2}` → match either of the words specified (words can themselves be made of wildcards)

Examples

- `ls *txt` → list all files ending with txt in the current directory
- `cp results/test_[X-Z] .` → copy files (test_X, test_Y, test_Z) from results directory to current directory
- `ls bkp/* | grep -E '201[0-5]'` → prints names of files/folders in bkp directory matching 2010/2011/2012/2013/2014/2015
- `rm junk.???` → remove any file starting with 'junk.' and ending with exactly three characters (junk.abc will be deleted, but not junkabc or junk.abcd)

Brace Expansion

Comes in handy when dealing with long path arguments and numbered sequences. [brace expansion wiki](#) and [when to use brace expansion](#) for in depth analysis and usage

- `ls *{txt,log}` → list all files ending with txt or log in the current directory
- `cp ~/projects/adders/verilog/{half_,full_}adder.v .` → copy half_adder.v and full_adder.v
- `mv story.txt{,.bkp}` → rename story.txt as story.txt.bkp (use `cp` to create bkp file as well retain original)
- `mv story.txt{.bkp,}` → rename story.txt.bkp as story.txt
- `mv story{,_old}.txt` → rename story.txt as story_old.txt
- `touch file{1..4}.txt` → touch file1.txt file2.txt file3.txt file4.txt
- `rm file{1..4}.txt` → rm file1.txt file2.txt file3.txt file4.txt
- `echo story.txt{,.bkp}` → echo command is handy to display the expanded version, in this case 'story.txt story.txt.bkp'

Standard Input, Output and Error Handling

By default all results of a command are displayed on the terminal, which is the default destination for '**standard output**'. But often, one might want to save or discard them or send as input to another command. Similarly, inputs to a command can be given from files or from another command. Errors are special outputs generated on a wrong usage of command or command name. The ability to control and use these three special files is one of the most important and powerful features of Shell and Linux in general

- ▶ Input and Output redirection can both be specified in same command, in any order
- ▶ White spaces around the redirection operators `>`, `>>`, `|`, `<` are ignored by the shell

Output redirection

- **grep -i error report/*.log > error.log** → create new file, overwrite if file already exists
- **grep -i fail test_results_20mar2015.log >> all_fail_tests.log** → creates new file if file doesn't exist, otherwise append the result to existing file
- **ls -l | grep '^d'** → pipe redirects standard output of ls command to grep command as standard input
- **./script.sh | tee output.log** → display output on terminal as well as write to file
- **(ls ; help source) | wc** → grouping with () allows to redirect output of more than one command

Input redirection

- **wc < report.log** → report.log is read by the shell and given to wc, so wc will not print filename in output

Process Substitution

[Process Substitution](#) allows one to use output of commands to be used as input or output filenames. This is commonly used to re-direct more than one command I/O which is not possible using the pipe redirect. See [input and output process substitution examples](#)

- **comm -23 <(sort file1.txt) <(sort file2.txt)** → [Bash Process Substitution](#) No need to sort the files first!
Note that there should be no space between < and (

Error redirection

Unlike input and output redirections, error handling syntax can be different across various shells. For example, bash shell allows separate redirection of standard error and standard outputs. But tcsh allows either standard output or combined standard output and error redirection. The symbols are also different: **2>** for **bash** and **>&** for **tcsh**

- **abcdxyz 2> /dev/null** → **bash**, redirect error to /dev/null, a special file whose size is always zero (assuming no command or an alias named abcdxyz). No message is displayed on terminal
- **./script.sh > output.log 2> error.log** → **bash**, redirect output to output.log and error to error.log separately, no message is displayed on terminal
- **./script.sh > output.log >& output_and_error.log** → **tcsh**, redirect output to output.log and combined output and error to output_and_error.log, no message is displayed on terminal

Variables

Two types – environment and shell variables. [Behavior varies with different type of shells](#) . A detailed look at bash variables [here](#).

- ▶ [difference between shell and environment variables](#)

Environment Variables

Environment variables are available system wide akin to global variables. Examples:

- HOME → home directory
- SHELL → path of default shell
- PATH → path(s) from where command is searched to be executed
- ▶ **printenv** → displays names and values of environment variables
- ▶ **echo \$HOME** → displays the value of HOME environment variable, note the use of \$

Shell Variables

Shell variables are effective only in the current shell. Example:

- HISTFILESIZE, HISTSIZE, HISTCONTROL, HISTFILE → command history related shell variables
- **set** → display the names and values of shell variables

User Defined variables

User can define variables for temporary use as well. Using lowercase is preferred to avoid conflict with shell or environment variables. Example:

- `dec2bin=({0..1}{0..1}{0..1}{0..1}{0..1}{0..1}{0..1}{0..1}); echo ${dec2bin[2]}` → prints 00000010.

Read more on this brace expansion [here](#). Note that use of ; allows multiple commands on same line

Command Substitution

A useful feature, especially in scripting, is command substitution. Somewhat similar to the pipe operator which redirects the output of one command to another, command substitution helps to redirect command output as input to another command or assigned to a script variable. Please avoid using deprecated back ticks `` syntax

- `rm -r $(cat remove_files.txt)` → remove all files/folders mentioned in separate lines of remove_files.txt
 - `cat remove_files.txt | xargs rm -r` → equivalent command
- `sed -r -i "s/(.*)$(pwd | xargs basename) \1/" *.txt` → use of double quotes allows to use output of other commands within \$(). In this example, the current directory name is inserted at start of every line
- Further reading [here](#)

Process Control

Process

Process is any running program. Program is of course a set of instructions written to perform a task

Job

In Shell parlance, job is a process that is not a [daemon](#), i.e job is an interactive program with user control. Daemons, to simply put, are background processes

Useful Commands

ps

- Lists current processes
- First column indicates the process id (PID) of a job
- `-e` → option to display all processes
- `-f` → option to display processes with detailed information
- `kill -9 PID` → kill command is used to terminate a process by using the PID number
- `jobs -l` → shell built-in command to display jobs launched from current shell

top

- Display real time information on all processes including those by other users of the machine
- Press `M` (uppercase) to sort the processes by memory usage
- Press `q` to quit the command
- Tip: Press `W` (uppercase) to write your favorite view of top command to .toprc file and quit immediately, so that next time you use top command, it will display in the format you like
- [htop](#) is better/prettier alternative to top, install instructions [here](#)

free -h

- Display amount of free and used memory in the system, `-h` option shows the numbers in human readable format

pgrep

- Search for processes based on name and other attributes like username. `-l` option lists process name also

bjobs

- Lists all processes launched by user on connected servers
- `bkill PID` → to terminate such processes using PID

Suggested Reading

- [Linux Processes](#)
- [Job Control commands](#)
- [Useful examples for top command](#)
- [Managing Linux Processes](#)
- [Process Management](#)

Running jobs in Background

Often commands and scripts can take more than few minutes to complete, but user might still need to continue using the shell. Opening a new shell might not serve the purpose if local shell variable settings are needed too. Shell provides the & operator to push the command (or script) execution to background and return the command prompt to the user. However, the standard outputs and errors would still get displayed on the terminal unless appropriately redirected

- **tkdiff** **result_v1.log** **result_v2.log** & → tkdiff, if installed, shows differences between two files in a separate window. If & is not used, the program would hog the command prompt

Pushing current job to background

What if you forgot the & operator, program started running and killing it might corrupt lot of things? Fret not, Shell has got it covered

1. Press **Ctrl+z** → suspends the current running job
2. **bg** → bg command would push the recently suspended job to background
3. Continue using shell
4. **fg** → fg command would bring the recently pushed background job to foreground

Customizing Shell

Through use of aliases, functions, shell variables, etc one can change certain shell behaviors. Changing them through interactive shell is usually valid only for current session. To make permanent changes, shell provides default file names to put the settings and also ways to use custom file names, prevent reading default files, etc

man bash - under the heading **INVOCATION**

“When an interactive shell that is not a login shell is started, bash reads and executes commands from /etc/bash.bashrc and ~/.bashrc, if these files exist. This may be inhibited by using the --norc option. The --rcfile file option will force bash to read and execute commands from file instead of /etc/bash.bashrc and ~/.bashrc.”

Recommended set up for bash customization

Purpose	File
Customize shell variables	~/.bashrc
Readline initialization	~/.inputrc
Aliases and functions, call the file from .bashrc	~/.bash_aliases
History of commands	~/.bash_history

Note that these files differ between different shells. For example, .tcshrc for **tcsh** shell

.bashrc

Some distros like Ubuntu come with ~/.bashrc already created with useful configurations. (If the file is

modified, one can see the original at `/etc/skel/.bashrc`) See more details about what the `.bashrc` file does [here](#). Apart from setting shell variables, calling aliases file, etc one can also add commands to be executed everytime a new interactive shell is opened. Examples:

- **shopt -s autocd** → As the name implies, one can change directory just using the directory name, without having to explicitly type the **cd** command, provided it doesn't clash with command names, aliases, etc More on [shopt](#) built-in command
- **source ~/.bash_aliases** → source allows to execute commands from a file. In this case, it allows use of separate file to store aliases

.inputrc

[Readline](#) library allows user to interact/edit command line. By default Emacs style short-cuts is enabled, which can be changed to vi style if preferred. Key-bindings and readline settings are typically put in `~/.inputrc`

- **"\e[A": history-search-backward** → up arrow to match history starting with partly typed text
- **"\e[B": history-search-forward** → down arrow to search in forward direction
- **"\C-d": unix-filename-rubout** → Ctrl+d to delete from cursor backwards to filename boundary
- **set echo-control-characters off** → turn off control characters like ^C(Ctrl+C) from showing on screen
- **set completion-ignore-case on** → ignore case for Tab completion
- **set show-all-if-ambiguous on** → combines single and double Tab presses behavior into single Tab press
- [Simpler introduction to readline](#)

.bash_history

By default, history commands are stored in `~/.bash_history`, can be changed using `HISTFILE` variable. These settings are typically placed in `~/.bashrc`

- **HISTSIZE=5000** → for simpler concept, this variable affects how many commands are in history of current shell session. Use negative number for limitless size
- **HISTFILESIZE=10000** → for simpler concept, this variable affects how many commands are stored in the history file. Use negative number for limitless file size
- **shopt -s histappend** → append to history file instead of overwriting
- **HISTCONTROL=ignorespace:erasedups** → don't save commands with leading space (a good way to prevent command you do not wish to save) and erase all previous duplicates matching current command line. Read more [here](#)
- [common history across sessions](#)

.bash_aliases

To avoid cluttering `~/.bashrc` file, it is recommended to store aliases and functions in separate file (`~/.bash_aliases` for example). Before creating an alias or function, use **type your_alias_name** to avoid overriding existing command/alias/etc. Using **alias** command without argument on terminal will show all aliases currently set

- **alias c='clear'** → alias **clear** command to just the single letter c. Note that there should be no space characters around = symbol
 - Similarly, **p='pwd'**, **e='exit'**, **h='history'**, etc
- **alias b1='cd ../'** → alias b1 to go back one hierarchy above. Similarly, one can use b2, b3, b4, b5, etc
- **alias app='cd /home/xyz/Android/xyz/app/src/main/java/com/xyz/xyzapp/'** → alias frequently used long paths. Particularly useful when working on multiple projects spanning multiple years. And if aliases are forgotten over the years, they can be recalled by opening the alias file or using alias command
- **alias oa='gvim ~/.bash_aliases'** → meta aliasing! Short-cut to open aliases file with your favorite editor

- **alias** sa='source ~/.bash_aliases' → useful to apply new additions to current session
- **alias** ls='ls --color=auto' → colorize output, auto option intelligently turns on color only when standard output is connected to terminal, so that color coding isn't passed along with output when re-directing to another command or file
 - **alias** l='ls -ltrh' → map your favorite options, plus color output as previously set alias will be used
- **alias** grep='grep --color=auto' → colorize line numbers, matched pattern, etc
- **alias** s='du -sh * | sort -h' → sort files/directories by size and display in human-readable format
- **ch()** { **man** \$1 | **sed** -n "/^\s*\$2/,/^\$/p" ; } → use functions when dealing with arguments. Example use of this command help alias: **ch** ls -F, **ch** grep -o, etc
 - **ch()** { **what**is \$1; **man** \$1 | **sed** -n "/^\s*\$2/,/^\$/p" ; } → also prints description of command
 - [explain](#) command does a much better job than this alias
 - [explainshell](#) does similarly, but online
- **\ls** → override alias and use original command by using the \ prefix

Suggested Reading

- [Sensible bash customizations](#)
- [shell config subfiles](#)
- [command line navigation](#)

Emac mode Readline Short-cuts

- **Ctrl+c** → kill a misbehaving command (actually interrupts the command by sending abort signal, some commands may not respond, in that case use Ctrl+z to suspend, find PID using ps/pgrep/top etc and kill -9 PID to terminate the command)
- **Ctrl+c** → also used to abort the currently typed command and give fresh command prompt
- **Ctrl+z** → suspends the current running job
- **Tab** → the tab key completes the command (even aliases) or filename if it is unique, double tab press gives list of possible matches if it is not unique
- **Ctrl+r** → Search command history. After pressing this key sequence, type characters you wish to match from history, then press Esc key to return to command prompt or press Enter to execute the command
- **Esc+b** → move cursor backward by one word
- **Esc+f** → move cursor forward by one word
- **Esc+Backspace** → delete backwards upto word boundary
- **Ctrl+a** or **Home** → move cursor to beginning to prompt
- **Ctrl+e** or **End** → move cursor to end of command line
- **Ctrl+l** → preserve whatever is typed in command prompt and clear the terminal screen
- **Ctrl+u** → delete from beginning of command line upto cursor
- **Ctrl+k** → delete from cursor to end of command line
- **Ctrl+t** → swap the previous two characters around. For ex: if you typed sp instead of ps, press Ctrl+t when the cursor is to right of sp and it will change to ps
- **Esc+t** → swap the previous two words around
- **!\$** → last used argument (ex: **cat temp.txt** followed by **rm !\$**)
 - **Esc+.** → will insert the last used argument, useful when you need to modify before execution
- Mouse scroll button click → highlight text you want to copy and then press scroll button of mouse in destination to paste the text
 - **xinput set-button-map 11 1 0 3** → use this command to disable middle mouse paste (11 is mouse id, use **xinput list** to determine yours) **xinput set-button-map 11 1 2 3** to enable back

Linux: Shell Scripting

Scripting generally is a subset of programming language, where set of instructions are executed line by line (interpreted), without a compilation step in between. For example – C and JAVA are programming languages, whose programs are traditionally compiled to get an executable file and then run. On the other hand – Perl and Python are run by an interpreter. Apart from its choice use as interactive shell, **bash** is also widely used as default scripting language on Linux machines

Need for Scripting

- Automate repetitive manual tasks
- Create specialized and custom commands

Suggested Reading

- [Difference between scripting and programming languages](#)
- [Why Shell Scripting](#)
- [Bash guide, with downloadable pdf](#)
- [Ryan's tutorial on bash scripting](#)
- [Introduction to Bash Shell Scripting](#)
- [Bash Scripting Tutorial](#)
- [Common bash scripting issues faced by beginners](#)
- [Writing robust shell scripts](#)
- [Testing exit values in bash](#)

Elements of bash Scripting

Element	Description
#!/bin/bash	A script can specify the path of the interpreter as the first line of the script file
#comment	Single line comments start with the # character
echo	Print command to send text to standard output, \n is automatically added
read variable_name	Get input from user interactively
\$#	Number of command line arguments passed to the script
\$1, \$2, ... upto \$9	Variables holding command line arguments
\$0	Variable holding name of script
"\$@"	Contains all command line arguments as a string
\$?	Exit status of last command or script

Hello User

```
#!/bin/bash
#Print greeting message
echo Hello "$USER"
#Print day of week
echo Today is "$(date -u +%A)"
```

Hello User bash script

Let's start with the customary “Hello World” program. This being scripting language, let's change it a bit. The first line of the above code provides the path of the scripting program – bash in our case (Use **type bash** to

know the path in your environment). The ! character following # in the first line makes it a special comment to specify the path of interpreter and is necessary for any script to work as intended. The 2nd and 4th line are comments. The two echo commands highlight a very important aspect of shell programming – the seamless use of commands within the script. Note the use of double quotes - \$USER will get replaced with its value while printing and \$(date -u +%A) will be treated as command substitution. If single quotes had been used instead, the strings would be interpreted literally – do change it and execute the script to understand better.

```
> chmod +x hello_user.sh
> ls -l hello_user.sh
-rwxr-xr-x 1 guest guest 104 Mar 12 17:17 hello_user.sh
> ./hello_user.sh
Hello guest
Today is Thursday
> █
```

Executing hello_user.sh bash script

To run the script, we need to first give the execute permission (**chmod +x hello_user.sh**). And then use **./hello_user.sh** to run – dot represents current directory, to avoid conflict with any other potential same script name lying in paths defined by PATH environment variable.

- ▶ **.sh** is the customarily used extension for shell scripts, not a compulsory syntax
- ▶ Using a text editor, such as **vim**, is better than using the cat command for writing scripts

Command Line Arguments

```
#!/bin/bash
# Print line count of files given as command line argument
echo No of lines in "$1" is "$(wc -l < "$1")"
echo No of lines in "$2" is "$(wc -l < "$2")"
```

Using command line arguments

Command line arguments are saved in positional variables starting with \$1, \$2, \$3, etc upto \$9. The arguments are separated by white spaces. If a particular argument requires multiple word string, enclose them within single or double quotes. The special variable \$0 saves the name of the script itself (which is useful to make the script behave differently based on name used)

When more than 9 arguments are needed, the built in shell command **shift** can be used – each invocation will overwrite \$1 with \$2, \$2 with \$3 and so on

In the example given, the script simply prints the line count of files given as argument while invoking the script. For example, **./command_line_args.sh report.log error.log**

Accepting Interactive arguments

```
#!/bin/bash
# quotes can be ignored when there are no substitutions
echo Hi there! This script returns the sum of two numbers
# -p option allows to set a prompt for user input
read -p 'Enter first number: ' number1
read -p 'Enter second number: ' number2
# use double quotes for substitutions
echo The sum of "$number1" and "$number2" is: "$((number1 + number2))"
# use single quotes for printing special shell characters like )
echo 'Thank you for using the script, Have a nice day :)'
```

Interactively accept user arguments

Variables and Comparisons

- `dir_path=/home/guest` → space has special meaning in bash, cannot be used around `=` in variables
- `greeting='hello world'` → use single quotes to club special characters
- `user_greeting="hello $USER"` → use double quotes for substitutions
- `echo "$user_greeting"` → use `$` when variable's value is needed
- `num=534` → numbers can also be declared
- `((num = 534))` → but using `(())` for number variables and arithmetic makes life much easier
- `[[-e story.txt]]` → test if the file/directory exists
- `[[$str1 == $str2]]` → for string comparisons (note use of `$`)
- `((num1 > num2))` → for comparing number variables (use of `$` is optional)

Suggested Reading

- [bash arithmetic expressions](#)
- [difference between test, \[and \[\[](#)
- [bash FAQs](#)

if-then-else

```
#!/bin/bash
# Check if number of arguments is 2
# [[ ]] are keywords in bash, improved version of [ ] command
# -ne stands for 'not equal to' comparison
# white-space is required by syntax between arguments within [[ ]]
if [[ $# -ne 2 ]]
then
    echo Error!! Please provide two file names
    # By default, exit value is 0, and indicates successful execution
    # Changing exit values and using $? variable allows automated handling
of command/script execution
    # For this simple script, 1 indicates Error in script execution
    exit 1
else
    # Use ; to combine multiple instructions/commands in same line
    # -f option checks if file exists, ! negates the value
    if [[ ! -f $1 ]] ; then
        echo Error!! "$1" is not a valid file; exit 1
    else
        echo No of lines in "$1" is "$(wc -l < "$1")"
    fi
    # && operator executes the RHS command only if LHS command succeeds
    # Thus, the below two lines are equivalent to if-then-else structure
    [[ ! -f $2 ]] && echo Error!! "$2" is not a valid file && exit 1
    echo No of lines in "$2" is "$(wc -l < "$2")"
fi
```

Using if-then-else conditional execution

When handling user provided arguments, it is always advisable to check the sanity of arguments. A simple check can reduce hours of frustrating debug when things go wrong

for-loop

```
#!/bin/bash
# Ensure atleast one argument is provided
if [[ $# -eq 0 ]] ; then
    echo "Error!! Please provide atleast one file name" ; exit 1
else
    # while using variables, NEVER use white spaces
    # otherwise shell would treat it as a shell command
    # or, use (( )) to enforce arithmetic context
    ((file_count = 0))
    ((total_lines = 0))
    # $@ contains all the arguments as a string
    # every iteration, variable i gets next string
    for i in "$@"
    do
        [[ ! -f $i ]] && echo Error!! "$i" is not a valid file && exit 1
        echo Number of lines in "$i" is "$(wc -l < "$i")"
        ((file_count++))
        ((total_lines = total_lines + "$(wc -l < "$i")" ))
    done
    echo "Total Number of files is $file_count"
    echo "Total Number of lines is $total_lines"
fi
```

Using for-loop iterative control structure

```
#!/bin/bash
# C style for-loop
for((i = 0; i < 5; i++))
do
    echo $i
done
```

C style for-loop

while-loop

```
#!/bin/bash
# Print 5 4 3 2 1 0 using while loop
((i = 5))
# as a generic rule, use [[ ]] for strings/file tests
# and (( )) for arithmetic comparisons
while(( i >= 0 ))
do
    echo $i
    ((i--))
done
```

while-loop

Ensuring Error Messages on Terminal

As with shell commands, the output from scripts can be redirected to files. While this is desirable, it may be required to display error messages on terminal as well as allowing regular output to be redirected. This can be ensured using the special shell file **/dev/tty**

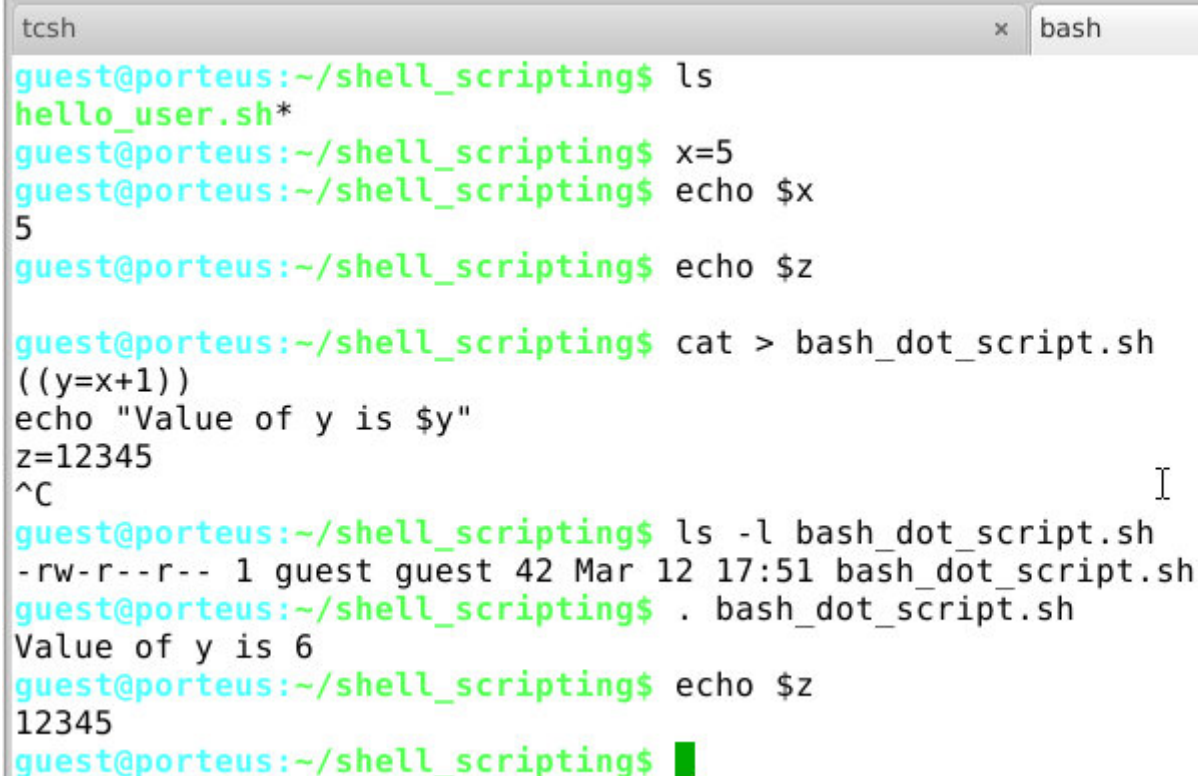

```
#!/bin/bash
# Ensure that Error message will always be displayed on terminal
# even if output of script is redirected to another file
# If redirection is not used, script behaves exactly
# same as without use of > /dev/tty
if [[ $# -eq 0 ]] ; then
    echo 'Error!! Please provide atleast one file name' > /dev/tty ;
exit 1
else
    for i in "$@"
    do
        [[ ! -f $i ]] && echo Error!! "$i" is not a valid filename >
/dev/tty && exit 1
        echo Number of lines in "$i" is "$(wc -l < "$i")"
    done
fi
```

Using /dev/tty to display error messages on current terminal

Using source command to run a bash script

Usually, scripts are run by giving execute permission to the script file and using the script name - similar to a command. What the shell does is, it creates a subshell to run the script. This results in variables, functions and aliases that may have been defined in the parent shell not available to the script. And those defined within the script are not visible to the parent shell. To overcome these restrictions – use `. script.sh` or `source script.sh`

- ▶ `.` is not available in shells like tcsh, use `source` instead
- ▶ The script language should be same as parent shell. For example, using `source bash_script.sh` in tcsh will most likely give errors as syntax is different between the two



```
tcsh x bash
guest@porteus:~/shell_scripting$ ls
hello_user.sh*
guest@porteus:~/shell_scripting$ x=5
guest@porteus:~/shell_scripting$ echo $x
5
guest@porteus:~/shell_scripting$ echo $z
guest@porteus:~/shell_scripting$ cat > bash_dot_script.sh
((y=x+1))
echo "Value of y is $y"
z=12345
^C
guest@porteus:~/shell_scripting$ ls -l bash_dot_script.sh
-rw-r--r-- 1 guest guest 42 Mar 12 17:51 bash_dot_script.sh
guest@porteus:~/shell_scripting$ . bash_dot_script.sh
Value of y is 6
guest@porteus:~/shell_scripting$ echo $z
12345
guest@porteus:~/shell_scripting$ █
```

Using . to run a bash script

As shown in the example above, local variable in parent shell (x) is available for the script and the variable declared in script (z) is available in parent shell after execution. Also note that, one need not add the `#!/bin/bash` line in the script, nor is the execute permission needed to run the script

Debugging script

```
guest@porteus:~/shell_scripting$ bash -xv hello_user.sh
#!/bin/bash
# Print greeting message
echo Hello "$USER"
+ echo Hello guest
Hello guest
# Print day of week
echo Today is "$(date -u +%A)"
date -u +%A) "
date -u +%A)
date -u +%A
++ date -u +%A
+ echo Today is Friday
Today is Friday
```

Debug options

- **-x** → expands sequence of command execution and values of variables
- **-v** → verbose, print script lines as they are read and executed
- **set -xv** → using this line immediately after `#!/bin/bash` is another way to enable debugging
- Very useful to see flow of control statements like loop and of course variable values
- [shellcheck](#) – online static analysis tool that gives warnings and suggestions
 - See [github](#) link for more info and install instructions