Course 4 - Hands-on Introduction to Linux Commands and Shell Scripting

Week 1

Introduction to Linux

Module 1: - Introduction to Linux

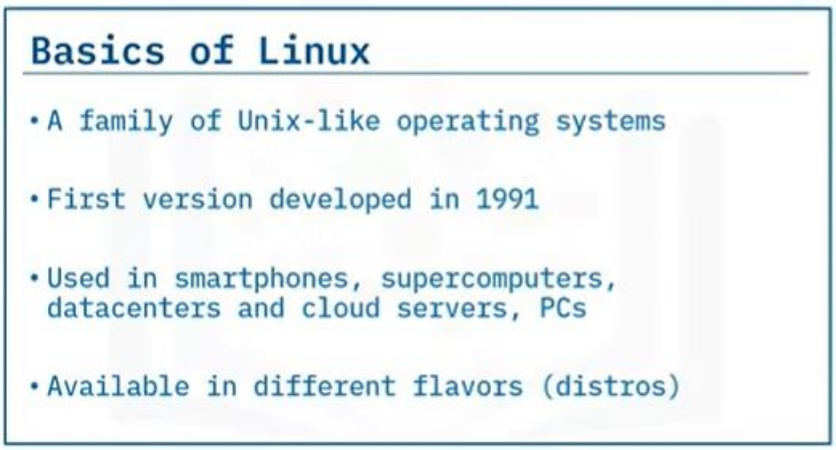
Course Introduction

Learning Linux commands and shell: -

Learning Linux commands and shell scripting makes you more flexible, and these are key skills whether you are a:

- Software Developer
- Data Scientist
- Data Engineer
- Systems Administrator
- DevOps Professional
- System Architect
- Software Engineer

Linux – basics, commands, and shell scripting: -



Getting information

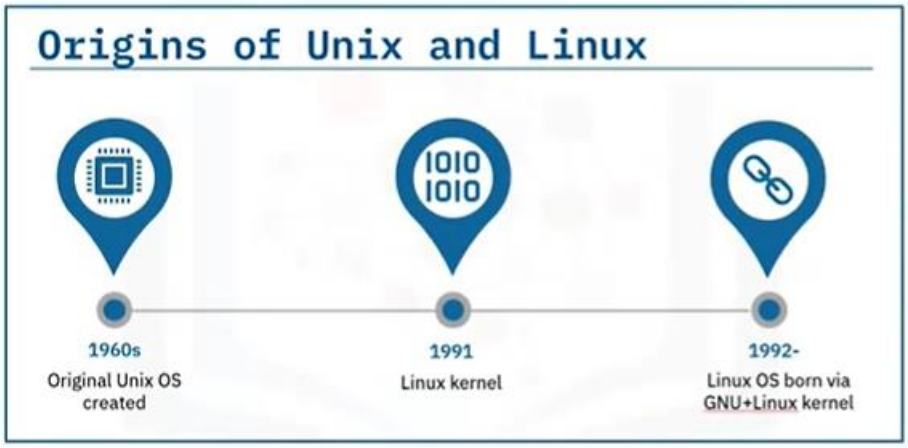
Some common shell commands for getting information include:

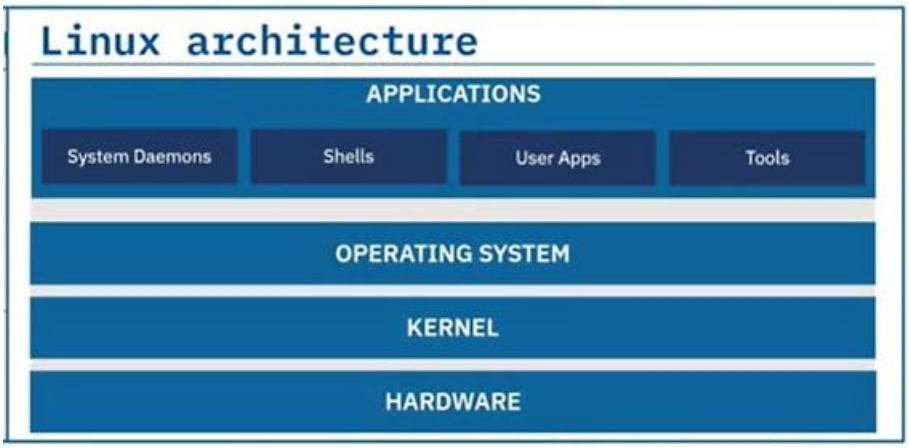
- whoami – username
- id – user ID and group ID
- uname – operating system name
- ps – running processes
- top – resource usage
- df – mounted file systems
- man – reference manual
- date – today's date

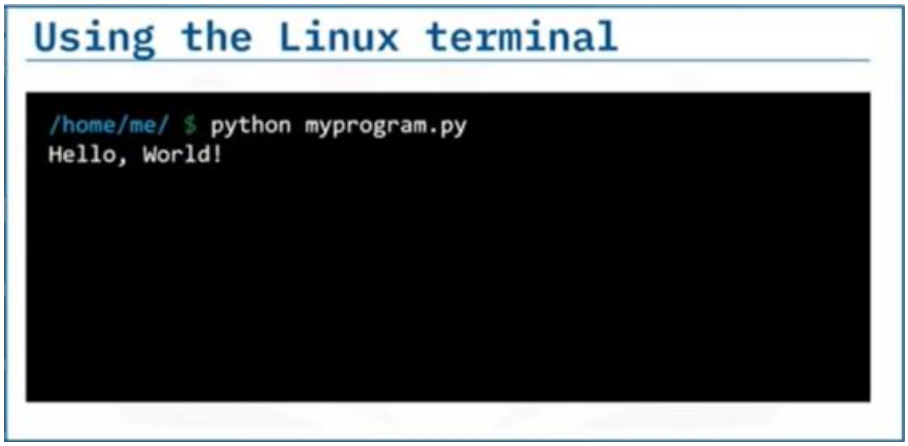
What is a shell?

- User interface for running commands
- Interactive language
- Scripting language

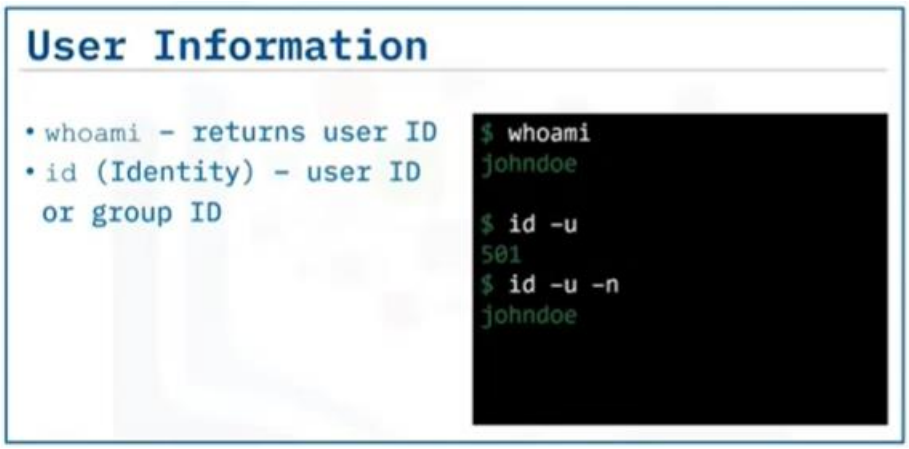
Introducing Linux: -

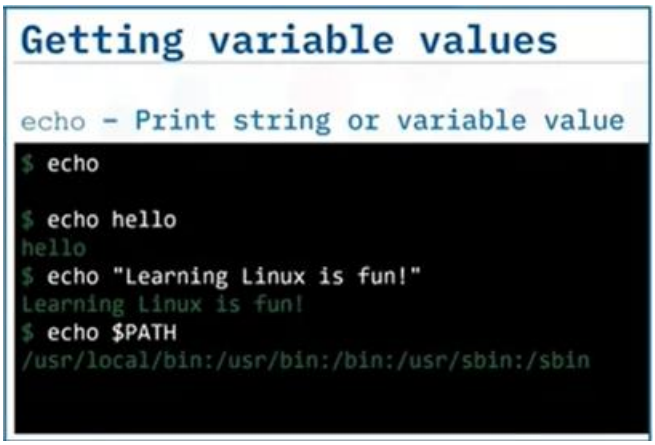




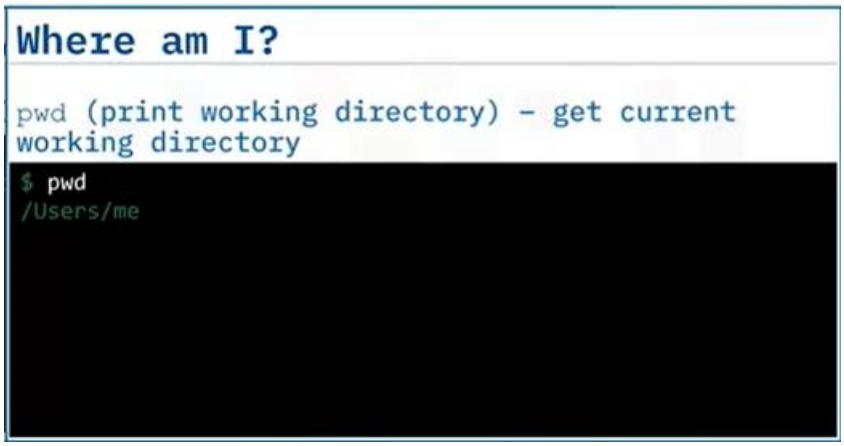


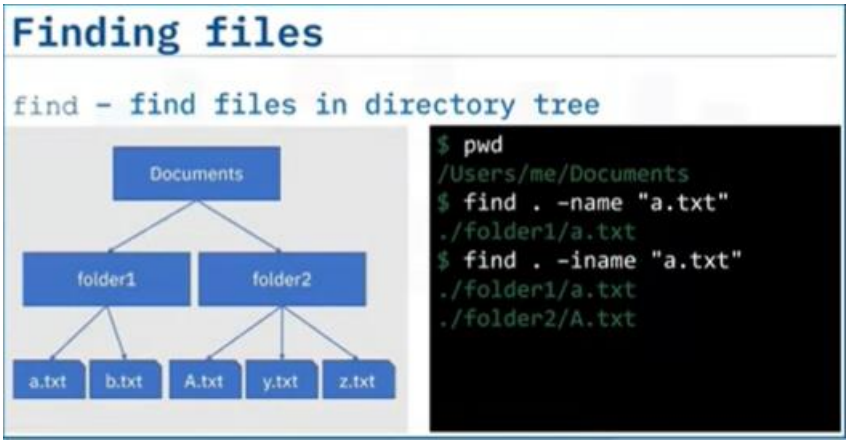
Informational Commands: -





File and directory navigation Commands: -

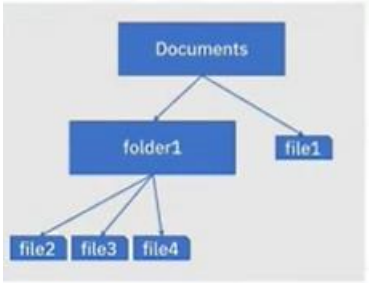




File and directory management commands: -

Removing files and directories

`rm` (remove) - Remove file or directory




```
$ pwd
/Users/me/Documents
$ ls
file1 folder1
$ rm file1
$ ls
folder1
$ rm folder1
rm: folder1: is a directory
$ rm -r folder1
$ ls

$ mkdir empty_folder
$ rmdir empty_folder
$ ls
```

Creating files

`touch` - Create empty file, update file date

```
$ pwd
/Users/me/Documents
$ touch a.txt b.txt c.txt d.txt
$ ls
a.txt b.txt c.txt d.txt
$ date -r notes.txt
Mon  8 Nov 2021 16:37:45 EST
$ touch notes.txt
$ date -r notes.txt
Fri 12 Nov 2021 10:46:03 EST
```

Copying files and directories

`cp` (copy) - Copy file or directory to destination

To copy files:

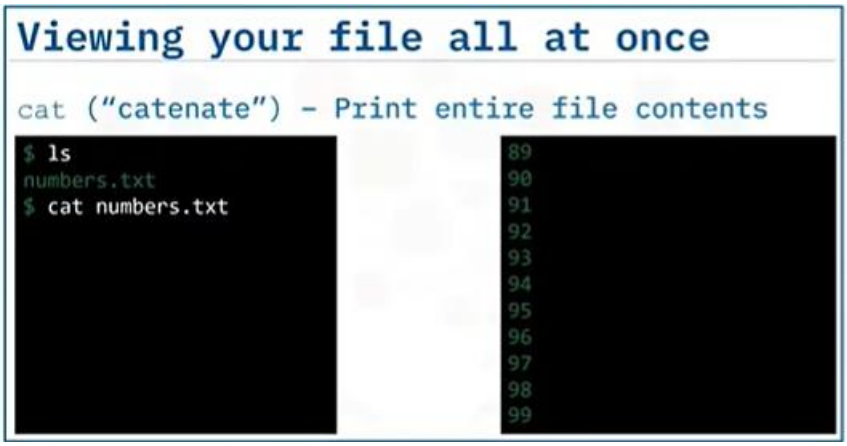
```
$ cp /source/file /dest/filename
$ cp /source/file /dest/
```

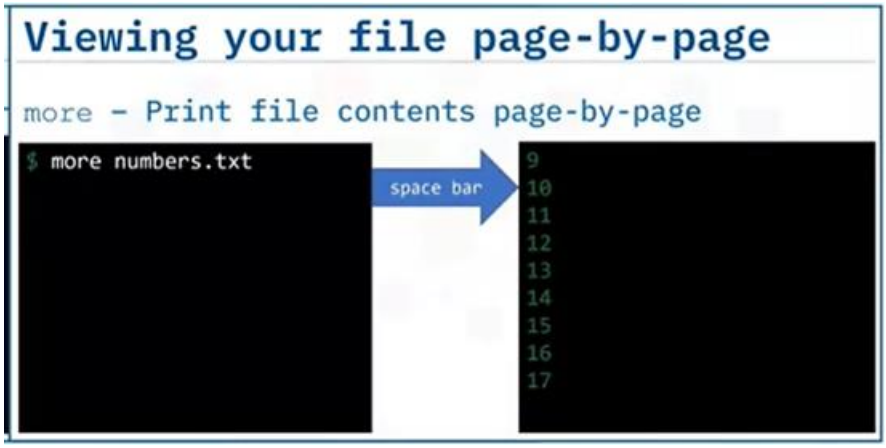
To copy directories:

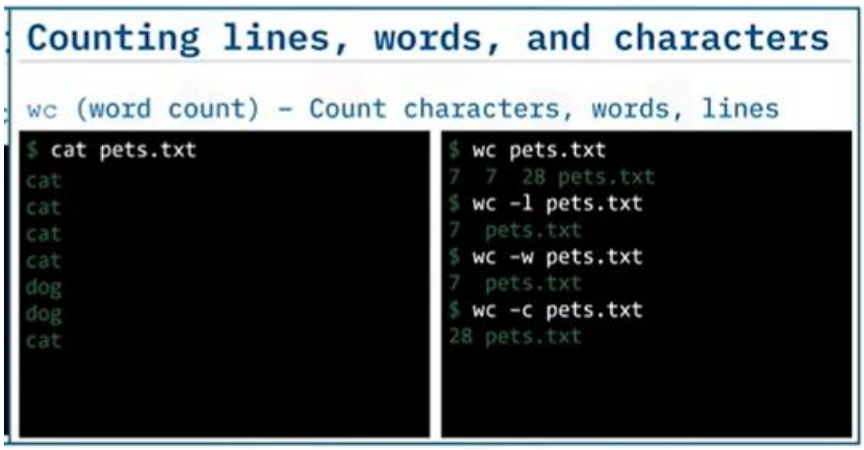
```
$ cp -r /source/dir/ /dest/dir/
```

```
$ ls
notes.txt Documents
$ cp notes.txt Documents
$ ls Documents
notes.txt
$ cp -r Documents Docs_copy
$ ls
notes.txt Documents Docs_copy
$ ls Docs_copy
notes.txt
```

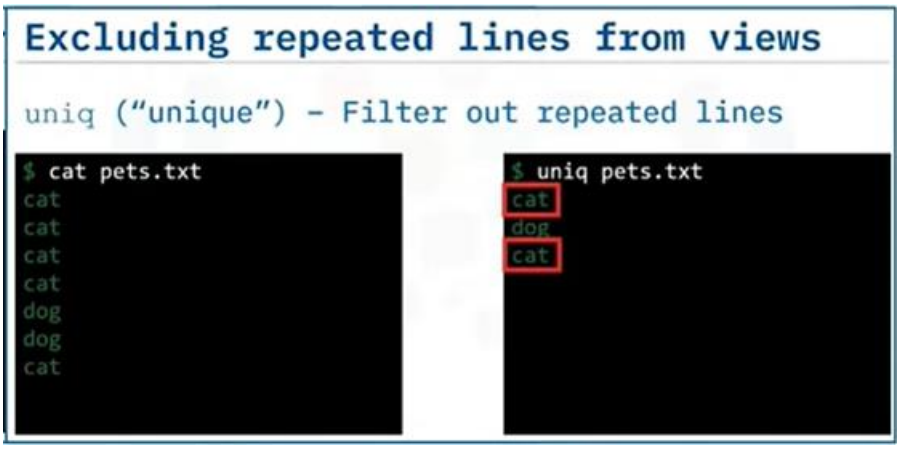
Viewing file content: -

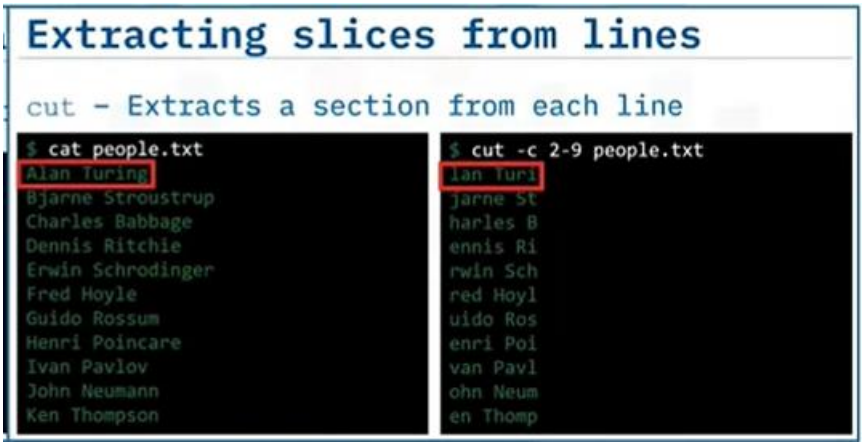




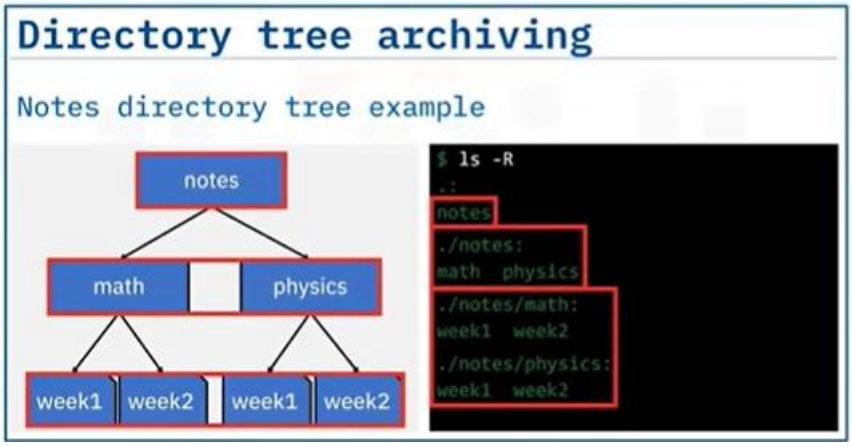


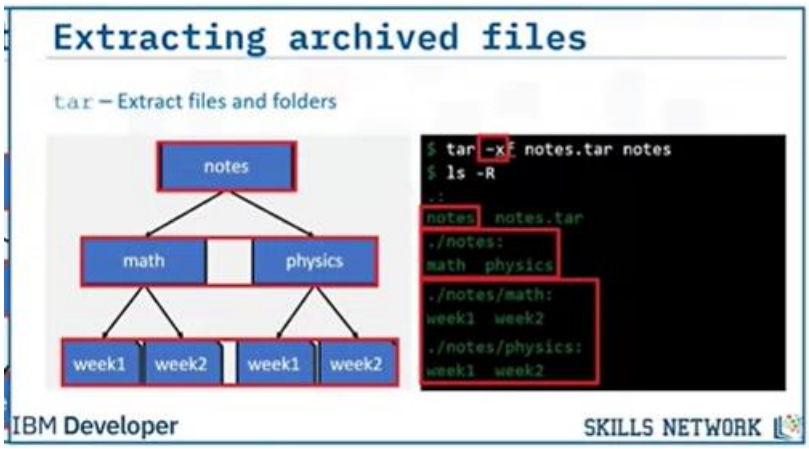
Customizing Views of file content: -

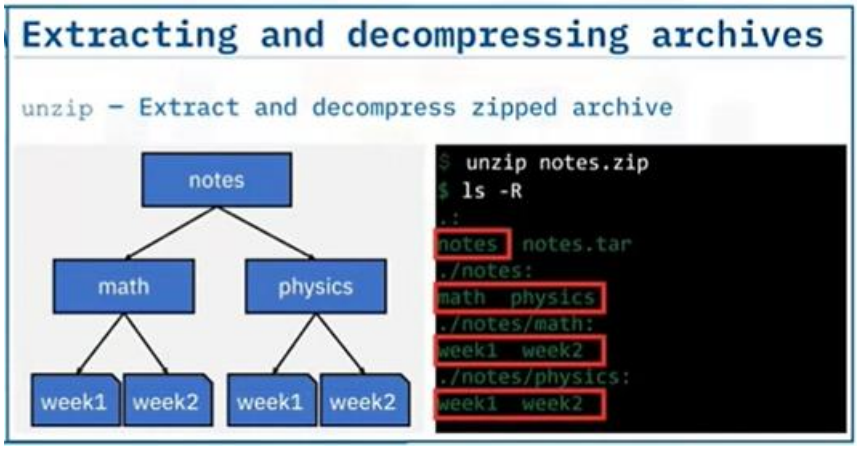




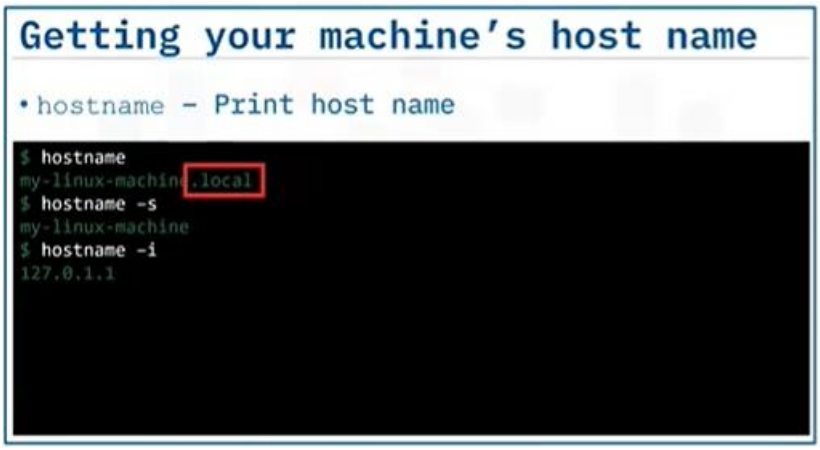
File archiving and compression commands: -

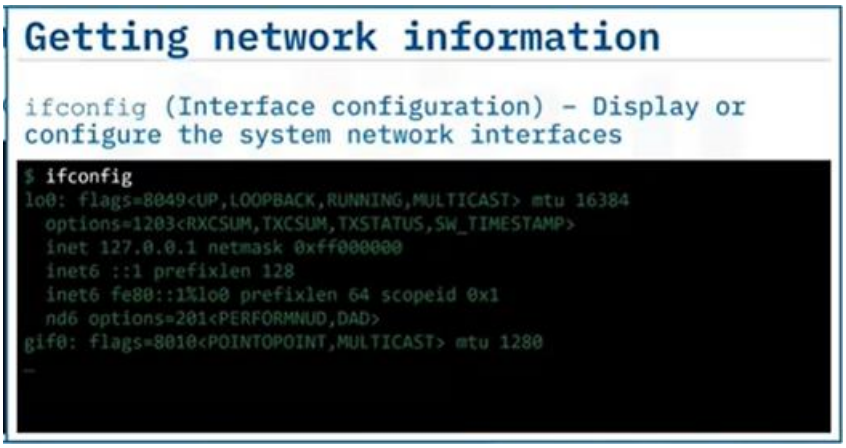


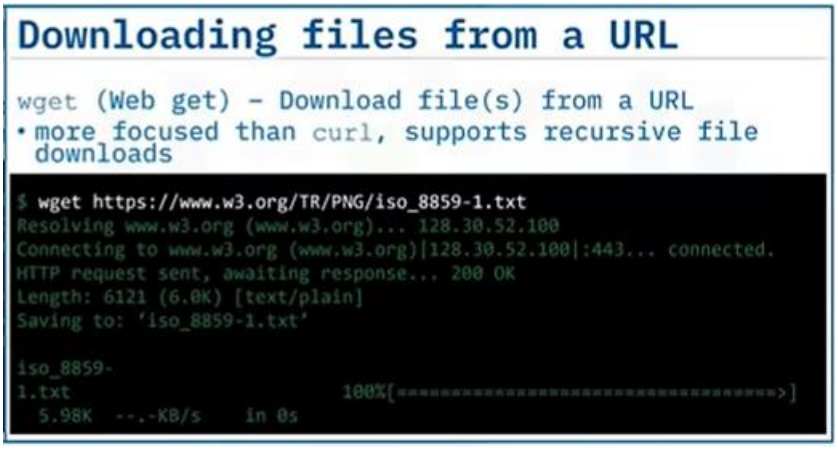




Networking commands: -







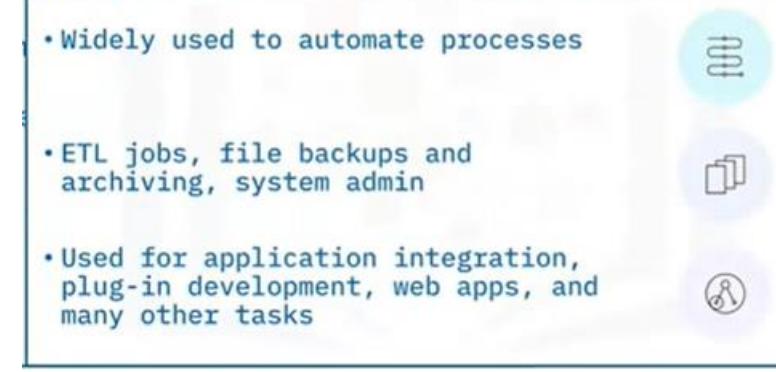
Shell scripting basics: -

What is a script?

- Script: list of commands interpreted by a scripting language
- Commands can be entered interactively or listed in a text file
- Scripting languages are interpreted at runtime
- Scripting is slower to run, but faster to develop

What is a script used for?

- Widely used to automate processes
- ETL jobs, file backups and archiving, system admin
- Used for application integration, plug-in development, web apps, and many other tasks



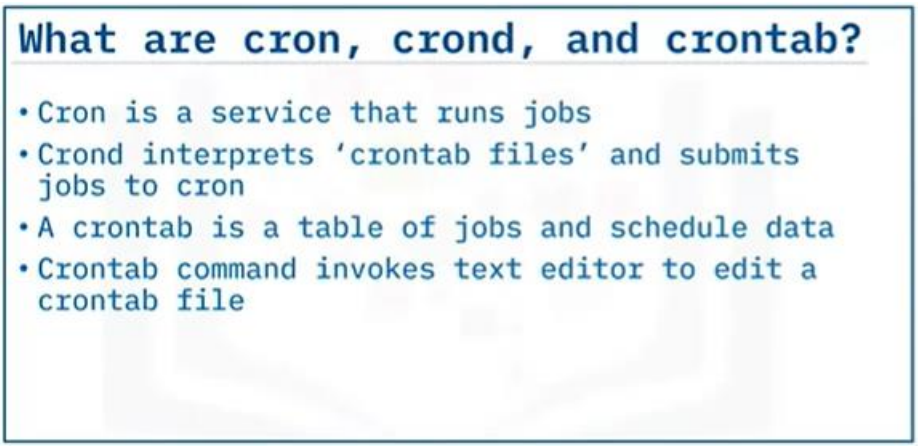
'Hello World' example shell script

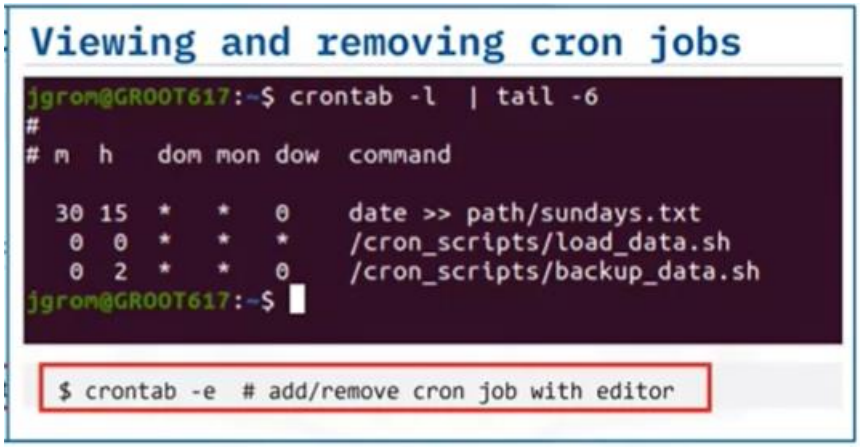
Create the shell script:

```
$ touch hello_world.sh
$ echo '#! /bin/bash' >> hello_world.sh
$ echo 'echo hello world' >> hello_world.sh
```

Filters, pipes, and variables: -

Scheduling jobs using cron: -





1] Introducing Linux and Unix

What is an Operating system?

An Operating system is software that:

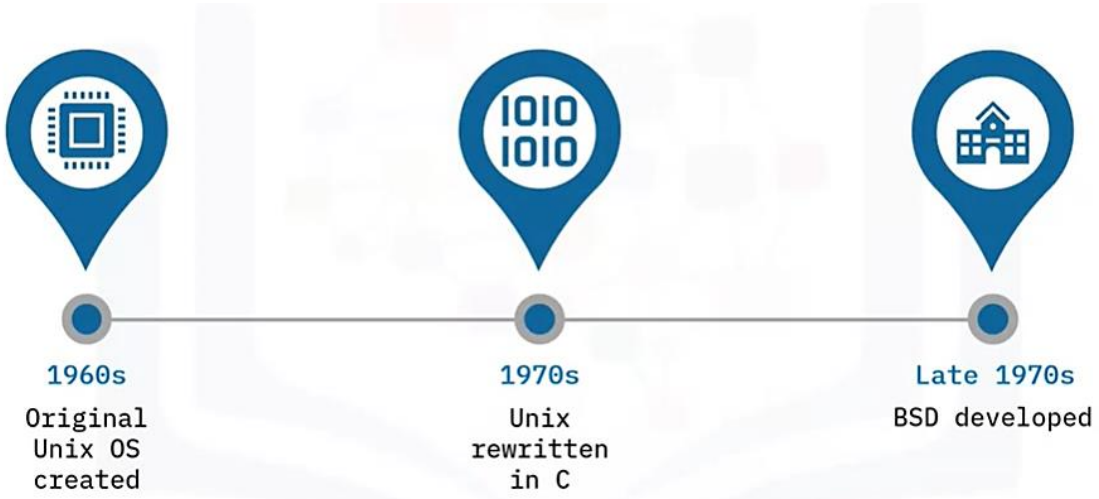
- Manages computer hardware and resources
- Allows interaction with hardware to perform useful tasks.

What is Unix?

- Unix is a family of operating systems.
- Popular Unix-based OSs include:
  - Oracle Solaris (and Open Solaris)
  - FreeBSD

- HP-UX
- IBM AIX
- Apple macOS

Unix beginnings: -



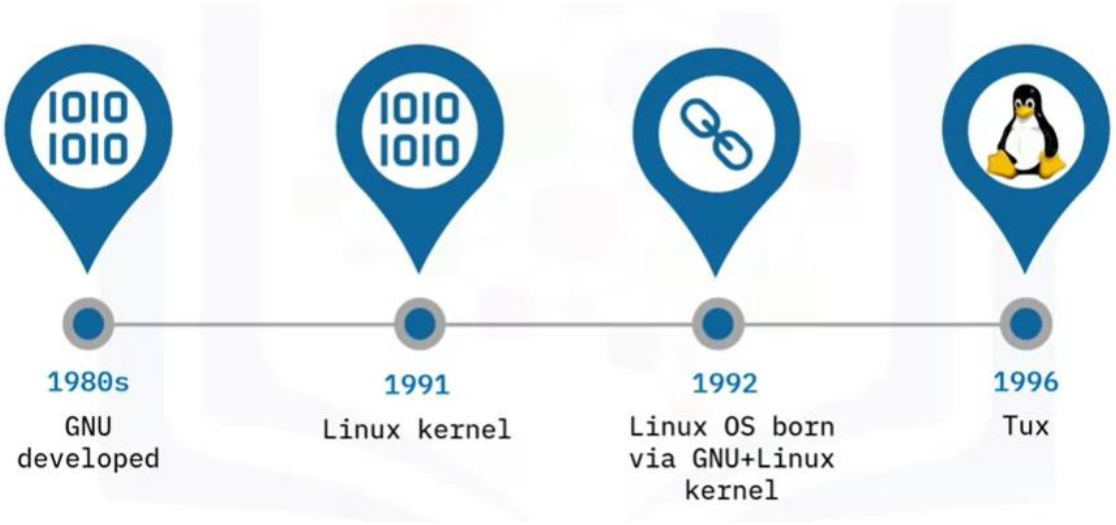
What is Linux?

- Linux is a family of Unix-like OSs
- Linux was originally developed as an effort to create a free, open source Unix OS.

Linux Features: -

- Free and open source
- Secure
- Multi-user
- Multitasking
- Portability

Linux beginnings: -




```
Date: 25 Aug 91 20:57:08 GMT Organization: University of Helsinki

From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:08 GMT Organization: University of Helsinki

Hello everybody out there using minix - I'm doing a (free) operating system (just
a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has
been brewing since april, and is starting to get ready. I'd like any feedback on
things people like/dislike in minix, as my OS resembles it somewhat (same
physical layout of the file-system (due to practical reasons) among other
things).
```

Today: -

- BSD-based macOS runs on millions of devices
- Billions of Linux instances run on servers, serving the modern web
- Modern Linux OSs are gaining popularity for PCs

Linux use cases today: -

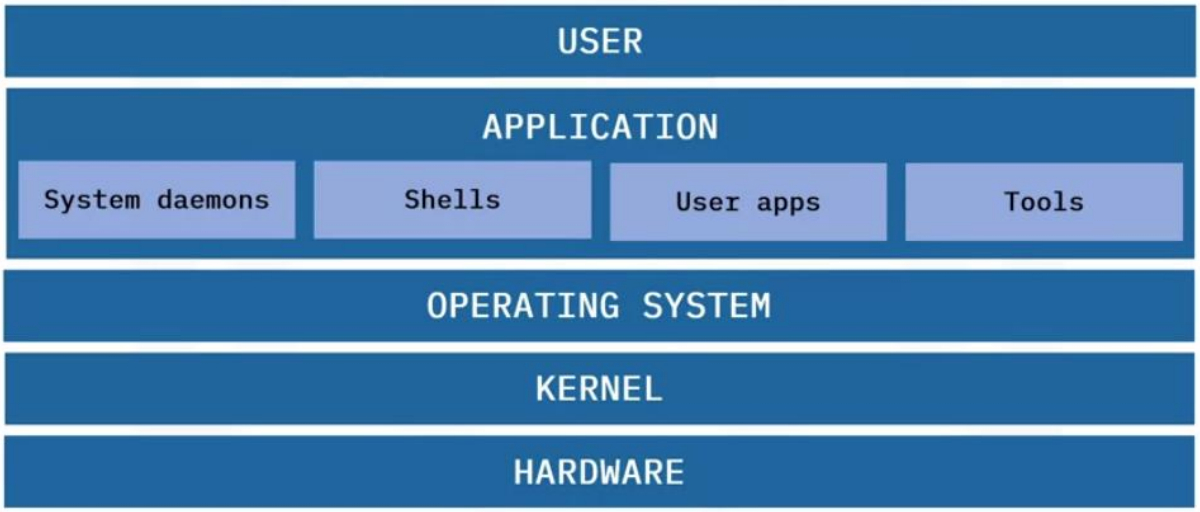
- Android
- Supercomputers
- Data centres and cloud servers
- PCs

Summary: -

- Unix is a family of operating systems dating from the 1960s
- Linux was originally developed in the 1980s as a free, open source alternative to Unix.
- Linux is multi-user, portable, and supports multitasking.
- Linux is widely used today in mobile devices, supercomputers, data centres, and cloud servers.

2] Overview of Linux Architecture

Overview: -



The **user** is the person using the Linux system. Users interact with the system via the **application layer** that includes system daemons, shells, user apps, and tools. The applications communicate with the **operating system** to perform tasks. The OS is responsible for jobs that are vital for system stability such as job scheduling and keeping track of time. All Linux operating systems are built on top of the Linux kernel, which performs the most vital lower-level jobs. The **kernel** is the core component of the operating system and is responsible for managing memory, processing, security, and so on. The kernel interacts with the **hardware layer**, which includes all the physical electronic devices in the computer such as processors, memory modules, input devices, and storage.

1. **User –**
    - The person using the Linux Machine
    - Completes tasks as:
        - Using a Web browser to send an email
        - Using a music player to listen to a favorite song

2. **Application Layer –**

   Any software that lets you perform a task:

   - System tools
   - Programming languages
   - Shells
   - User apps (browsers, text editors, games, and many more)

3. **Operating System –**

   - Controls the jobs and programs vital to health and stability
   - Also –
     - Assigns software to users
     - Helps detect errors and prevent failures
     - Performs file management tasks

4. **Kernel –**

   - Lowest-level software in system
   - Starts on boot and remains in memory
   - Bridge between apps and hardware
   - Key jobs:
     - Memory Management
     - Process Management
     - Device Drivers
     - System Calls and Security

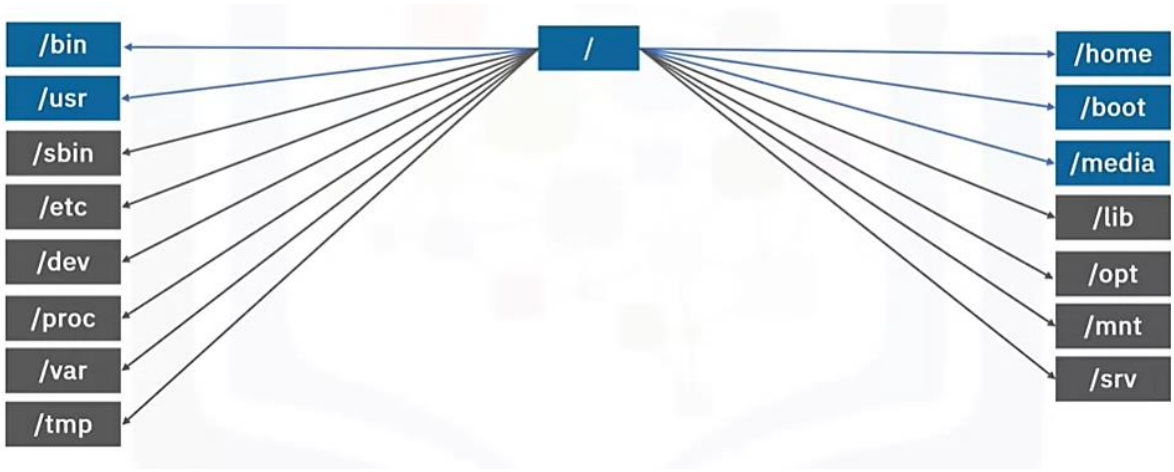
5. **Hardware –**

   Consists of all physical electronic devices on your PC:

   - CPU
   - RAM
   - Storage
   - Screen
   - USB devices

Linux filesystem: -

- Collection of files in your machine
- Begins at root directory (/)
- Tree-like structure
- Assigns appropriate access rights



- The very top of the Linux filesystem is the root directory, which contains many other directories and files.
- /bin, which contains user binary files. Binary files contain the code your machine reads to run programs and execute commands. It's called "slash bin" to signify that it exists directly below the root directory.
- /usr, which contains user programs
- /home, which is your personal working directory where you should store all your personal files
- /boot, which contains your system boot files, the instructions vital for system startup, and
- /media, which contains files related to temporary media such as CD or USB drives that are connected to the system.
- All the files and directories in a Linux system are organized into one of these designated folders, depending on the purpose of the file or directory.

Summary: -

- The Linux system consists of five key layers.
- The operating system runs on top of the Linux kernel and is vital for system health and stability.
- The kernel is the lowest-level software and enables applications to interact with your hardware.
- All system files are organized withing the Linux filesystem depending on their purpose.

3] Linux Distributions

What is a Linux distribution?

- A specific flavour of Linux OS.
- Also known as a distro.
- Uses the Linux kernel.
- Hundreds of Linux distros tailored for different audiences or tasks.

Linux distro differences: –

- System utilities
- GUI
- Shell commands
- Support types:
  - Community vs. enterprise
  - LTS vs. rolling release

Linux distros: Debian

- First release in 1993 (0.01), first stable release in 1996 (1.1)
- Stable, reliable, fully open source
- Supports many computers architectures
- Largest community-run distro

Linux distros: Ubuntu

- First release in 2004 (4.10)
- Debian-based
- Developed and managed by Canonical
- Three editions:
  - Ubuntu Desktop
  - Ubuntu Server
  - Ubuntu Core

Linux distros: Red Hat Linux

- A core Linux distro
- Stable, reliable, fully open source
- Managed by Red Hat
- Red Hat Enterprise Linux (RHEL)

Linux distros: Fedora

- Supports many architectures
- Very reliable and secure
- Actively developed, large community
- Sponsored by Red Hat

Linux distros: SUSE Enterprise

- SUSE Linux Enterprise (SLE) available in two editions:
  - Server (SLES)
  - Desktop (SLED)
- Supports many architectures
- SUSE Package Hub
- Maintained by SUSE

Linux distros: Arch Linux

- Do-it-yourself approach
- Highly configurable
- Requires strong understanding of Linux and system tools
- Leading-edge software

Summary: -

- Linux distros can be differentiated by their user interfaces, their shell applications, and how the operating system is supported and built.
- The design of a Linux distro is catered toward its specific audience.
- Debian is highly regarded in the sever space for its stability, reliability, and for being open source.
- Red Hat Enterprise Linux, an IBM subsidiary, is focused completely on enterprise customers.
- SUSE Linux Enterprise supports many architectures, such as ARM for Raspberry Pi.

4] Linux Terminal Overview

Overview of Linux Shell –

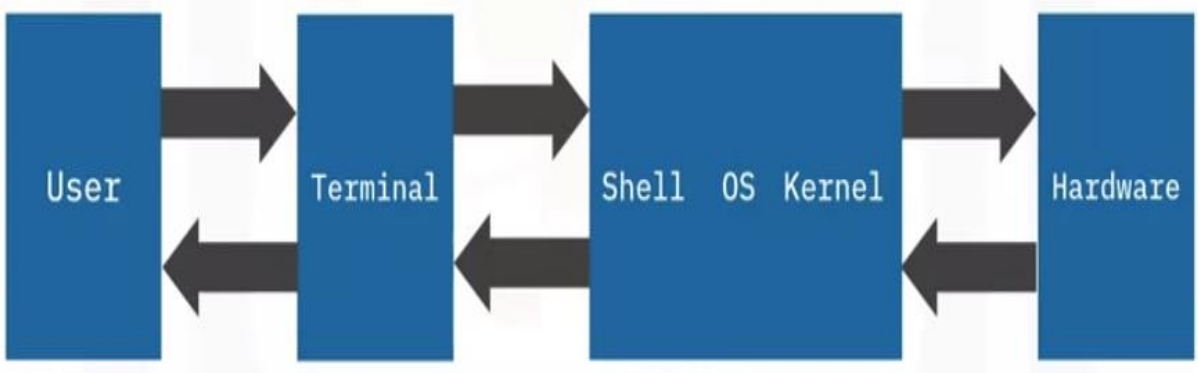
- The shell is an OS-level application that interprets commands:
  - Moving and copying files
  - Writing to and reading from files
  - Extracting and filtering data
  - Searching for data
- Shells:
  - Bash
  - Zsh

Overview of Linux Terminal –

- An application you can use to interact with the Linux shell
- Enter commands and receive output from them

```
/home/me/ $ python myprogram.py
Hello, World!
```

Communicating with Linux system –



How are commands run?

- First, we have a user who wants to run a command.
- They enter the command in a terminal, which is then relayed to the shell.
- The core components of the operating system and kernel translate the command for the hardware to perform.
- When the hardware completes the command, the kernel reads any changes or results and sends them back via the shell to the terminal for the user's information.
- The terminal is a powerful way to run applications and interact with your machine.

- Most terminals have a similar user interface for you to enter commands. The area where you enter commands is called the command line. And the vertical line, or cursor, is the command prompt. This indicates where the text that you type will be displayed.
- In this example, the current working directory is the Documents directory, which is inside the me directory, which is inside the home directory. The current working directory is the location where the shell will look for any commands that you specify to run, for example, the Python program in the previous example. Not all terminals display the full location, or path, of your current directory, so some will just display Documents here.

Paths in the Linux filesystem –

- Human-readable directory or file location
  /home/me/documents
- a/b – the file or directory named b inside the directory named a
- Special paths:
  - ~ Home directory
  - / Root directory
  - .. Parent of current directory
  - . Current directory

Changing the current working directory –

```
/home $ cd /
/ $ cd bin
/bin $ ./ls
[    cat   cp    dash    dd    echo    expr    kill
...
ls  mv   ps     rm        sh    stty      tcsh    unlink
/bin $ cd ~
/home/me $ ls
Documents  Pictures Downloads Movies  Music Desktop
```

```
/home $ cd ..
/ $ cd /media/my-usb-drive
/media/my-usb-drive $ cd ../../home/me/Documents
/home/me/Documents $ cd ..
/home/me $ python ./myprogram.py
Hello, World!
/home/me $
```

Summary: -

- A Linux shell is an OS-level application that you can use to enter commands and view the output of those commands.
- You use a terminal to send commands to the shell.
- You can use the cd command to navigate around your Linux filesystem.

Linux Terminal Tips - Tab completion, command history

Tab Completion

Many shells support a tool called "Tab Completion"

Tab completion allows a shell to auto-complete a command you're typing.

Support you're in your home directory ~, which contains:

- Pictures
- Videos
- Documents
- Downloads

And suppose your Documents folder *only* contained the folder:

- python-examples

Ex 1 - Tab Completion:
If you type:

`~ $ cd P`
and press TAB, the shell will autocomplete this to:

`~ $ cd Pictures/`
Because the Pictures directory is the only directory within your current folder that starts with a "P".

Ex 2 - Tab Completion for Long Path:
If you type:

`~ $ cd Do`
and press TAB, nothing will happen.

Because your current directory contains more than one directory that starts with "Do".

On the other hand, if you type:

`~ $ cd Doc`
and press TAB, the shell will autocomplete this to:

`~ $ cd Documents/`
If you press TAB again, the shell will autocomplete this to:

`~ $ cd Documents/python-examples/`
Because the folder python-examples is the only existing file within the ~/Documents directory.

Command history

Command history allows you to navigate previous commands you enetered using the arrow keys

Let's say you've entered the following commands:

~ $ **cd** ~/Documents/**python**-examples
~/Documents/**python**-examples $ **python3** myprogram.**py**
Hello, World!
~/Documents/**python**-examples $ **cd** /
/ $
You can run a previous command, by pressing the up-arrow key

Ex 1 - Running the last command:
If you press UP once:

~ $ cd ~/Documents/python-examples
~/Documents/python-examples $ python3 myprogram.py
Hello, World!
~/Documents/python-examples $ cd /
/ $ cd /
The shell will automatically put the last command you entered

~ $ cd ~/Documents/python-examples
~/Documents/python-examples $ python3 myprogram.py
Hello, World!
~/Documents/python-examples $ cd /
/ $ cd /

Ex 2 - Running previous command from session
If you had pressed UP *three times* instead of once:

~ $ **cd** ~/Documents/**python**-examples
~/Documents/**python**-examples $ **python3** myprogram.**py**
Hello, World!
~/Documents/**python**-examples $ **cd** /
/ $ **cd** ~/Documents/**python**-examples
It would automatically place the command you ran three commands ago (cd ~/Documents/python-examples)

After pressing enter, youwould find yourself back in your cd ~/Documents/python-examples directory:

~ $ cd ~/Documents/python-examples
~/Documents/python-examples $ python3 myprogram.py
Hello, World!
~/Documents/python-examples $ cd /
/ $ cd ~/Documents/python-examples
~/Documents/python-examples $

5] Creating and Editing Text Files

Popular text editors –

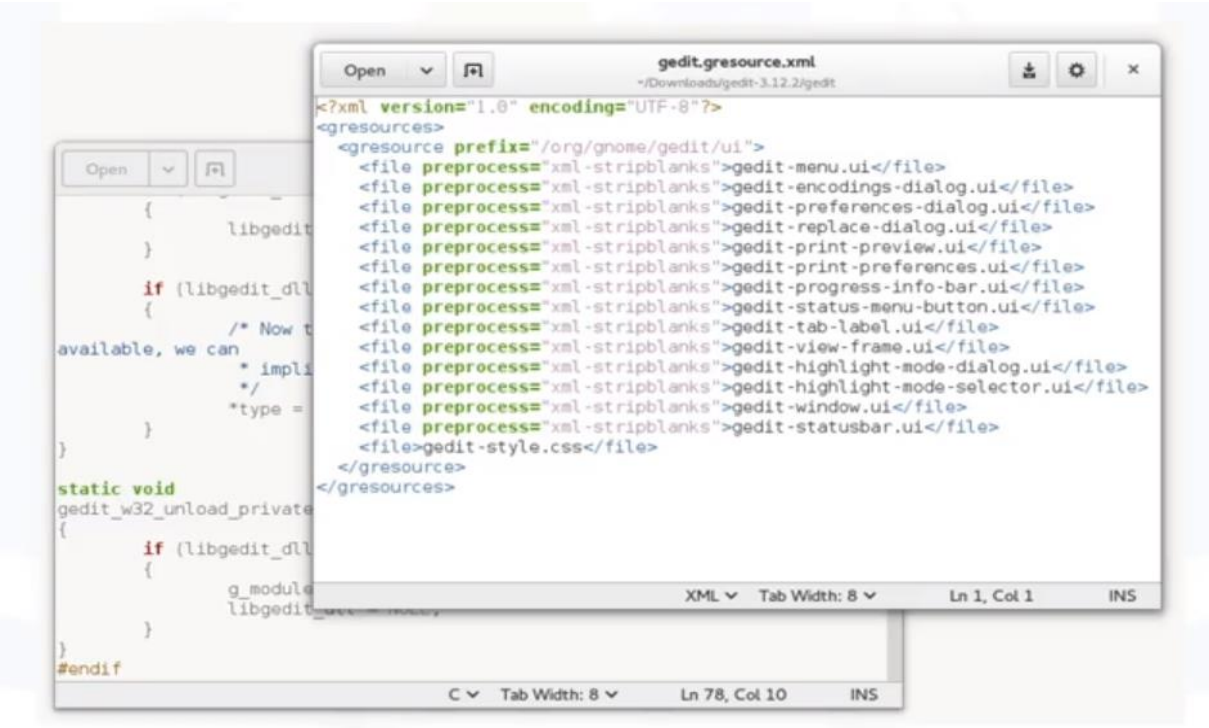
- Command-line text editors:
  - GNU nano
  - vi
  - vim
- GUI-based text editors:
  - gedit
- Command-line or GUI:
  - emacs

Features of gedit –

A general-purpose editor, easy to use with a clean, simple GUI:

- Integrated file browser
- Undo and redo
- Search and replace
- Extensibility

Syntax color coding



Features of GNU nano –

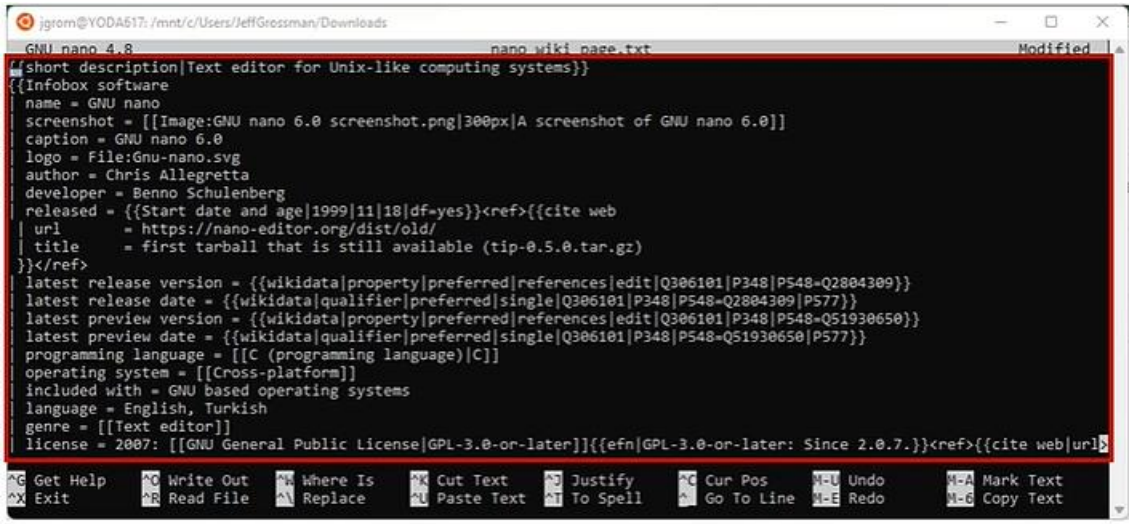
A command-line text editor providing:

- Undo and redo
- Search and replace
- Syntax highlighting
- Indenting groups of lines
- Line numbers
- Line-by-line scrolling
- Multiple buffers

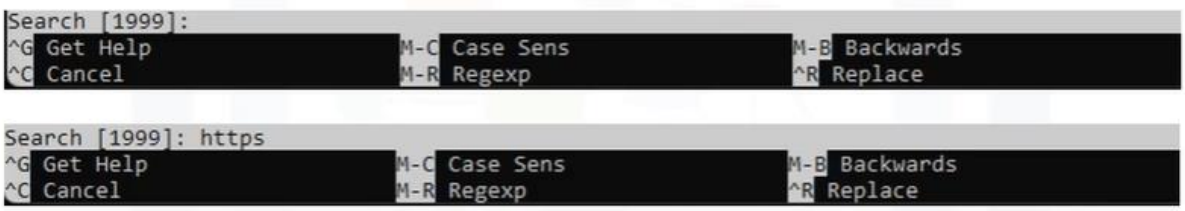
Using the GNU nano text editor –

To open a text file in GNU nano from the command prompt, type: -

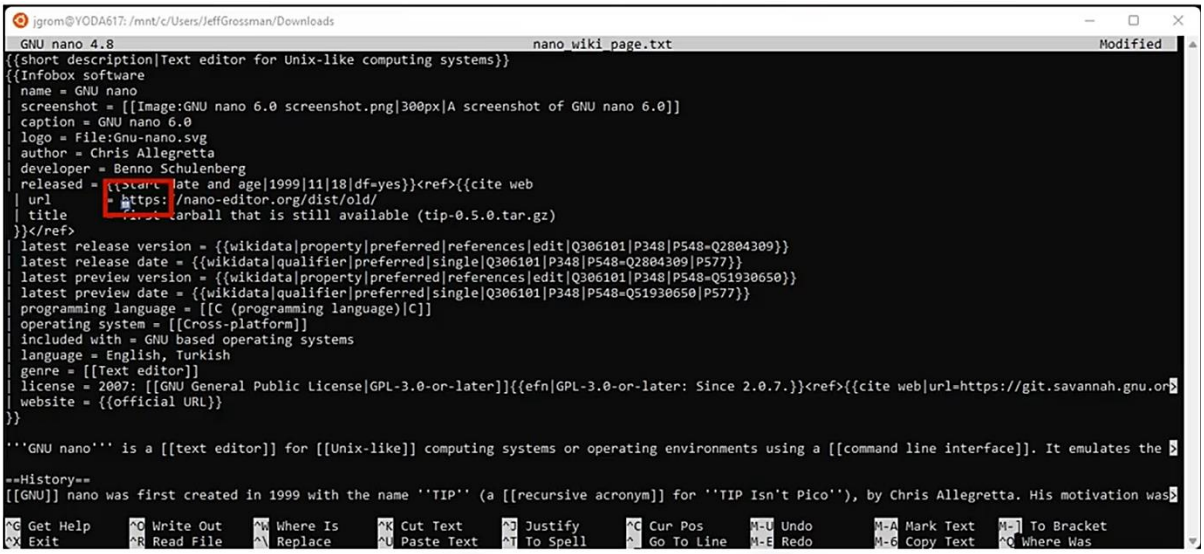
nano <filename>



Using nano to search for a string – Where Is



For example, to Get Help, press the ‘control’ and ‘G’ keys. Let’s see how to use a few of the editing options. To search for a text string, you can press ‘control W’ to use the ‘Where Is’ option. This opens a new pane at the bottom of the app window. Here’s a closer view of that new pane. Within the square brackets you can see the most recently searched string, which here is 1999. Type the string you want to find, for example ‘https’, and press Enter.



File editing with vim –

- Vim is traditional and very powerful command-line editor.
- To start vim, type: -
  vim
- To specify a file to edit, type: -
  vim <filename>
- Two basic modes:
  Insert and Command
- **Insert mode:**
  - Press i to enter Insert mode
  - Type some text
  - Press ESC to exit Insert mode and switch to Command mode
  - The text is written to the buffer at the cursor location.
- **Command mode:**
  - Enter **:sav example.tx**t to create a file and write the buffer to the file
  - Enter **:w** to write the buffer to the file without existing.
  - Enter **:q** to quit vim session
  - Enter **:q!** to quit without saving

Summary: -

- You can use a variety of command-line or GUI-based text editors to work with your Linux code.
- gedit is a GUI-based editor that provides many features to simplify your work.
- GNU nano is a command-line editor that provides similar functionality in a command-line format.
- And vim is another command-line editor that uses Insert mode to enter data and Command mode to work with the file.

6] Installing Software and Updates

Packages and package managers –

- Packages:
  - Archive files
  - For installing new software or updating existing software.
- Package managers:
  - Manage the download and installation of packages
  - Available for different Linux distros
  - Can be GUI-based or command line tools

Deb and RPM packages –

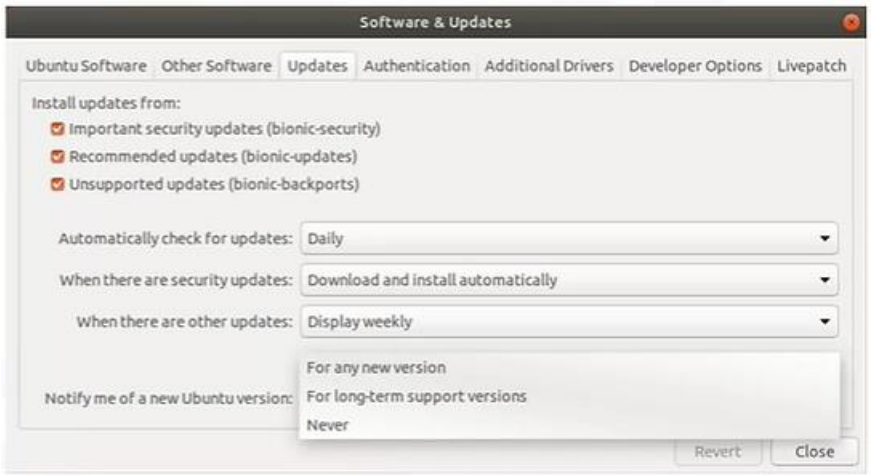
- Packages for Linux OS
- Distinct file types for different Linux OSs
- .deb files:
  - For Debian-based distributions such as Debian, Ubuntu, and Mint
  - deb stands for Debian
- .rpm files:
  - For Red Hat-based distributions such as CentOS/RHEL, Fedora, and openSUSE
  - RPM stands for Red Hat Package Manager.
- deb and RPM formats are equivalent
- If a package is only available in one format you can use alien to convert it:
- RPM to deb
  alien <package-name>.rpm
- deb to RPM
  alien -r <package-name>.deb

Package managers –

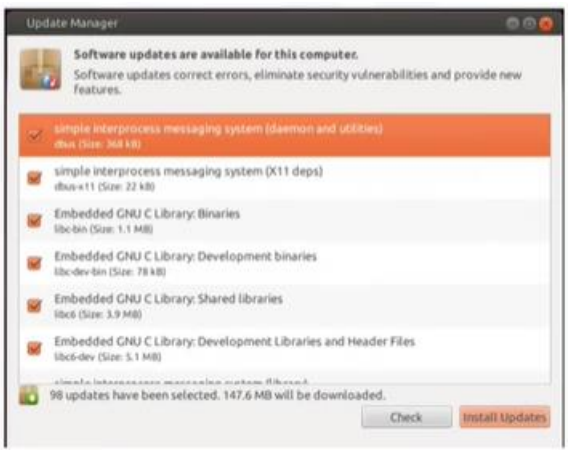
- Benefits:
  - Automatically resolve dependencies
  - Notify you when updates are available
  - GUI-based package managers can automatically check for updates.
  - Automatic or manual installation
- Linux distro package managers include Package Kit and Update Manager.

Updating deb-based Linux –

- GUI tool: Update Manager
  - Automatically checks for updates at configurable intervals.
  - Supports manual checking for updates.



- When available software updates are listed:
  - Select the updates you want to install
  - Click 'Install Updates'
  - Enter your user password and click ok
  - Installs updates in the background while you work



Command line: apt

```
$ sudo apt update
Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-backports InRelease [108 kB]
…
Get:27 http://security.ubuntu.com/ubuntu focal-security/universe amd64 c-n-f Metadata [14.1 kB]
Fetched 7935 kB in 2s (4746 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
50 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

```
$ sudo apt upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
…
After this operation, 12.1 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu focal/main amd64 libpython3.7-stdlib amd64 3.7.13-1+focal1 [1788 kB]
…
Get:14 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu focal/main amd64 libpython3.7-minimal amd64 3.7.13-1+focal1 [597 kB]
Preconfiguring packages ...
…
Preparing to unpack .../00-rsync_3.1.3-8ubuntu0.2_amd64.deb ...
Unpacking rsync (3.1.3-8ubuntu0.2) over (3.1.3-8ubuntu0.1) ...
…
Unpacking python3.7 (3.7.13-1+focal1) over (3.7.12-1+focal2) ...
…
Processing triggers for systemd (245.4-4ubuntu3.15) ...
Processing triggers for man-db (2.9.1-1) ...
```

Updating RPM-based Linux –

GUI tool: PackageKit

- Notifies you when updates are available
- Lists available software updates
- Installs updates in the background while you work

Command line tool: yum

```
$ sudo yum update
[sudo] password for user:
Fedora Modular 35 - x86_64 - Updates            3.1 MB/s | 2.8 MB     00:00
Last metadata expiration check: 0:00:01 ago on Fri 25 Mar 2022 09:08:11 PM EDT.
Dependencies resolved.
================================================================================
Package                         Arch   Version              Repo     Size
================================================================================
Upgrading:
…
kernel                          x86_64 5.16.16-200.fc35     updates  85 k
…
python3                         x86_64 3.10.3-1.fc35        updates  26 k
Transaction Summary
================================================================================
Install   30 Packages
Upgrade  752 Packages
Total download size: 1.2 G
Is this ok [y/N]:
```

```
$ sudo yum update
…
Is this ok [y/N]: y
Downloading Packages:
…
(605/782): python3-3.10.3-1.fc35.x86_64.rpm     408 kB/s |  26 kB     00:00
…
--------------------------------------------------------------------------------
Total                                           29 MB/s | 1.2 GB     00:41
Running transaction
…
 Upgrading     : python3-3.10.3-1.fc35.x86_64          425/1535
…
Upgraded:
…
 python3-3.10.3-1.fc35.x86_64
…
Complete!
```

Installing new software –

- **Installing a deb package with apt:**

  sudo apt install <package-name>

- **Installing an RPM package with yum:**

  sudo yum install <package-name>

Other software package managers –

- Python package managers include pip and conda
- Installing the pandas library:

```
pip install pandas

Collecting pandas
Downloading pandas-1.4.1-cp38-cp38-manylinux1_x86_64.whl (10.3 MB)
Requirement already satisfied: python-dateutil>=2.7.3 in
/usr/lib/python3/dist-packages (from pandas) (2.7.3)
Requirement already satisfied: pytz>=2017.2 in /usr/lib/python3/dist-
packages (from pandas) (2019.3)
Requirement already satisfied: numpy>=1.15.4 in /usr/lib/python3/dist-
packages (from pandas) (1.17.4)
Installing collected packages: pandas
```

Summary: -

- .deb and .rpm are distinct file types used by package managers in Linux operating systems
- deb and RPM formats can be converted from one to the other.
- Update Manager and PackageKit are popular GUI-based package managers used in deb- and RPM- based distros, respectively.
- apt and yum are popular command line package managers used in deb- and RPM-based distros, respectively.

Summary & Highlights:-

- Linux originated in the 1990s when Linus Torvalds developed a free, open source version of the Unix kernel.
- Linux is multi-user, portable, and supports multitasking.
- Linux is widely used today in mobile devices, desktops, supercomputers, data centers, and cloud servers.
- Linux distributions (also known as distros) differ by their UIs, shell, applications, and how the OS is supported and built.
- The design of a distro is catered toward its specific audience and/or use case.
- Popular Linux distributions include Red Hat Enterprise Linux (RHEL), Debian, Ubuntu, Suse (SLES, SLED, OpenSuse), Fedora, Mint, and Arch.
- The Linux system consists of five key layers: user, application, OS, kernel, and hardware.
- The kernel is the lowest-level software and it enables applications to interact with your hardware.
- The shell is an OS-level application for running commands.
- You use a terminal to send commands to the shell.
- You can use the cd command to navigate around your Linux filesystem.
- You can use a variety of command-line or GUI-based text editors such as GNU nano, vim, vi, and gedit.
- Deb and RPM packages contain software updates and installation files.
- You can use GUI-based and command-line package managers to update and install software on Linux systems.

Week 2

Introduction to Linux Commands

Module 1: - Introduction to Linux Commands

1] Overview of Common Linux Shell Commands

What is a shell?

- User interface for running commands
- Interactive language
- Scripting language

What is a shell?

- Default shell is usually Bash/bourne again shell.
- Many other shells, including sh, ksh tcsh. zsh, and fish.
- We will use Bash for this course.

```
$ printenv SHELL
/bin/bash
$ bash
```

```
$
```

```
>
```

Shell command applications –

- Getting information
- Navigating and working with files and directories
- Printing file and string contents
- Compression and archiving
- Performing network operations
- Monitoring performance and status
- Running batch jobs

Getting information –

Some common shell commands for getting information include:

- whoami – username
- id – user ID and group ID
- uname – operating system name
- ps – running usage
- top – resource usage
- df – mounted file systems
- man – reference manual
- date – today's date

Working with files –

Some common shell commands for working with files include:

- cd – copy file
- mv – change file name or path
- rm – remove file
- touch – create empty file, update file timestamp
- chmod – change / modify file permissions
- wc – get count of lines, words, characters in file.
- grep – return lines in file matching pattern.

Navigating and working with directories –

Very common shell commands for navigating and working with directories include:

- ls – list files and directories
- find – find files in directory tree
- pwd – get present working directory
- mkdir – make directory
- cd – change directory
- rmdir – remove directory

Printing file and string contents –

For printing contents or strings, common commands include:

- cat – print file contents
- more – print file contents page-by-page
- head – print first N lines of file
- tail – print last N lines of file
- echo – print string or variable value

Compression and archiving –

Shell commands related to file compression and archiving applications include:

- tar – archive a set of files
- zip – compress a set of files
- unzip – extract files from a compressed zip archive

Networking –

Networking applications include the following:

- hostname – print hostname
- ping – send packets to URL and print response
- ifconfig – display or configure system network interfaces
- curl – display contents of file at a URL
- wget – download file from URL

Running Linux on a Windows machine –

- Dual boot with a partition
- Install Linux on a virtual machine
- Use a Linux emulator
- Windows Subsystem for Linux (WSL)

Summary: -

- A shell is an interactive user interface for running commands, a scripting language, and an interactive language.
- Shell commands - used for navigating and working with files and directories.
- Shell commands can be used for file compression.
- curl and wget - display and download files from URLs.
- echo - prints string or variable values.
- cat and tail - used to display file contents.

2] Informational Commands

User Information –

- whoami – returns user ID
- id (Identity) – user ID or group ID

```
$ whoami
johndoe

$ id -u
501
$ id -u -n
johndoe
```

System Information –

uname (Unix name) – returns OS Information

- Kernal/os name, version number

```
$ uname
Darwin
$ uname -s -r
Darwin 20.6.0
$ uname -v
Darwin Kernel Version 20.6.0: Mon Aug 30 06:12:21 PDT 2021;
root:xnu-7195.141.6~3/RELEASE_X86_64
```

Displaying your disk usage –

- df (Disk Free) – Shows disk usage

```
$ df -h ~
Filesystem      Size  Used Avail Use% Mounted on
/dev/nvme0n1p2  2.0T  744G  1.2T  40% /home
```

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev             26G     0   26G   0% /dev
tmpfs           5.1G  2.6M  5.1G   1% /run
/dev/nvme1n1p5  255G   65G  177G  27% /
tmpfs            26G  223M   25G   1% /dev/shm
tmpfs           5.3M  4.1k  5.3M   1% /run/lock
tmpfs            26G     0   26G   0% /sys/fs/cgroup
/dev/loop2      230M  230M     0 100% /snap/gnome-3-34-1804/66
/dev/loop0      132k  132k     0 100% /snap/bare/5
/dev/loop1       59M   59M     0 100% /snap/core18/2128
…
```

Displaying current running processes –

- ps (Process status) – Running processes

```
$ ps -u root
 UID   PID TTY           TIME CMD
   0     1 ??         8:15.69 /sbin/launchd
   0    76 ??         0:13.27 /usr/sbin/syslogd
```

Monitoring system health and status –

top (Table of Processes) – Task manager

- Shows running tasks and their resource usage

```
$ top -n 3
PID    COMMAND          %CPU TIME      ...  USER      ...
38702  chrome           10.0 01:00.41  ...  johndoe   ...
38701  top              4.0  00:00.09  ...  johndoe   ...
38699  Spotify          3.0  01:00.07  ...  johndoe   ...
```

Getting variable values –

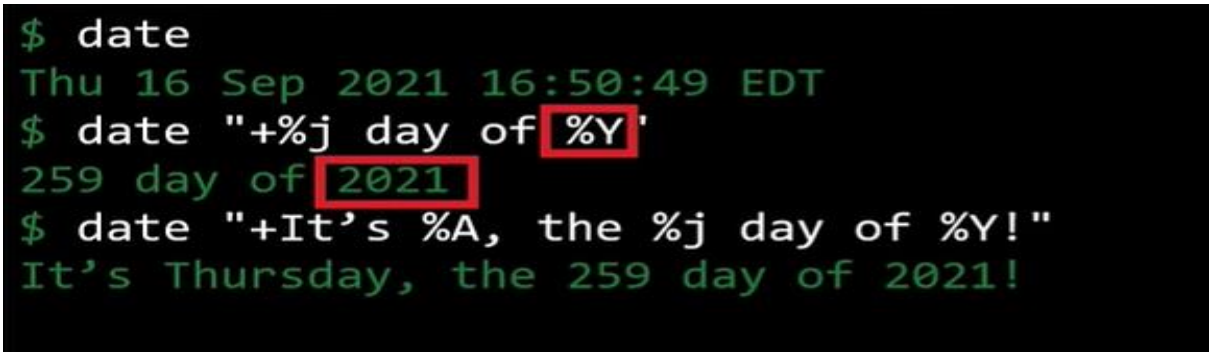
echo – Print string or variable value

```
$ echo

$ echo hello
hello
$ echo "Learning Linux is fun!"
Learning Linux is fun!
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

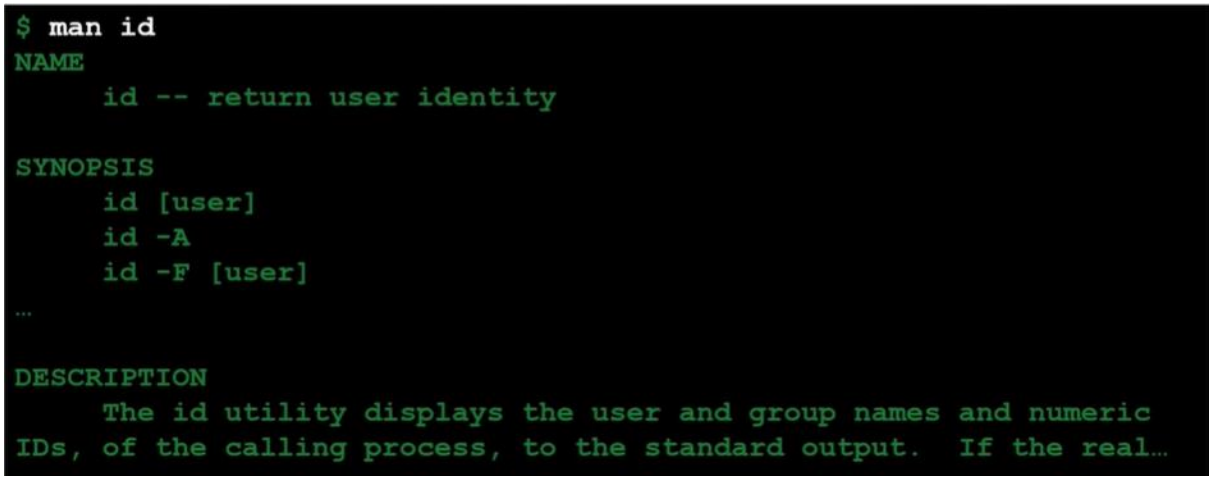
Getting date information –

date – Displays system date and time



Miscellaneous Information –

man (manual) – Shows manual for any command.



<u>Summary: -</u>

- Get user information with ***whoami*** and ***id***
- Check system disk usage using ***df***
- See process load using ***ps*** and ***top***
- Print string or variable values with ***echo***
- Use ***date*** to extract date information
- Find the manual for any command using ***man***

Reading: Getting Help for Linux Commands

Getting Help for Linux Commands

There is huge value in spending time exploring and experimenting with commands, but there are many things you can't figure out just by experimenting. You need to see what's out there, see what's possible, and learn where to look to get answers. Let's take a look at some great ways to get the information you need to help you make progress.

In this reading, you may see links to external sources. You can open them by right-clicking and pressing "Open in new tab."

1. Use the built-in man command

The man command, which stands for "manual", provides the standard way to access help for Unix-like commands from the command prompt. It has been in development since 1971.
You can get a listing of all the commands on your system that have a manual page by entering:

man -k .

The resulting list includes a brief description of what each command does.

To see the man page for a command, simply enter:

man command_name

All man pages are divided into several sections, including:

NAME

The name of the command or feature and a brief description of what it does.

SYNOPSIS

A summary of the command syntax, including any options and arguments that can be used.

DESCRIPTION

A more detailed description of the command, including its function and behavior.

OPTIONS

All the available options and arguments that can be used with the command.

EXAMPLES

Some examples of how to use the command.

SEE ALSO

Related commands and documentation that may be helpful.

You may also see other sections, including: EXIT STATUS, RETURN VALUE, ENVIRONMENT, BUGS, FILES, AUTHOR, REPORTING BUGS, HISTORY, and COPYRIGHT.

2. Install and use the `tldr` command

Similar to `man` pages, TLDR Pages is a free and open-source collaborative documentation effort. The goal is to create documentation that is more accessible than the traditional `man` pages, which tend to be quite verbose.
TLDR Pages, short for "Too Long; Didn't Read" and also known simply as *tldr*, provide examples for common use cases of various commands. The format of TLDR pages is similar to that of a cheatsheet.
You can install a command-line tool to access TLDR Pages from your terminal. Install it using the following command:

npm install -g tldr

Once you've installed the tool, you can use the `tldr` command to easily access the TLDR page of a command.

tldr command_name

The tool will display a short, easy-to-understand summary of the command along with some examples of how to use it.

3. Search Stack Overflow

Stack Overflow is a popular community-driven question and answer platform for programmers, developers, and system administrators. It has a vast repository of questions and answers related to various programming languages, tools, and operating systems, including Linux.

To search for information about commands on Stack Overflow, you can use the search bar on the homepage and enter the name of the command you're looking for, along with any specific keywords or parameters. You can also refine your search by adding relevant tags, such as "linux" or "command-line".

Once you've entered your search query, Stack Overflow will display a list of relevant questions and answers that match your query. You can browse through the results to find the information you need, and even post your own question if you can't find an answer to your specific query.

When searching for information about commands on Stack Overflow, it's important to check the date of the answers to ensure that the information is still current and relevant. You should also read through the comments and discussion threads to get a better understanding of the context and any potential issues or limitations related to the command you're researching.

Newest questions on Stack Overflow tagged "Linux": https://stackoverflow.com/questions/tagged/linux

4. Search Stack Exchange

Stack Exchange is a network of question and answer communities, similar to Stack Overflow, but covering a broader range of topics beyond just programming. There are several Stack Exchange communities that specialize in topics related to Linux and open source software, such as Unix & Linux, Ask Ubuntu, and Server Fault.

Visit the relevant community to search for information on Stack Exchange. Like Stack Overflow, you can use the search bar to enter the name of the command you're looking for, along with any keywords or parameters.

Unix and Linux community on Stack Exchange: https://unix.stackexchange.com/

5. Just google it!

Google is a powerful tool that can provide you the answer to almost any question. Learn how to enter the right queries and filter your results, such as by including "Wikipedia", "Stack Overflow", or "Linux" as part of your search. However, use at your own risk. Never blindly trust what you find on the web - there's a lot of noise out there!

6. Use the cheat sheets from this course

Throughout this course, you will encounter "cheat sheets" that condense the information you've learned into easy-to-reference guides. They are great for reviewing the material you've learned and can also help you out with your graded assignments.

7. Refer to Wikipedia's list of Unix commands:

Finally, Wikipedia maintains a list of commands that can be found on Unix operating systems, along with a short description. You can check the page to quickly reference a Unix command: https://en.wikipedia.org/wiki/List_of_Unix_commands

3] File and Directory Navigation Commands

Listing your directory contents –

ls (list) – list files and directories

```
$ ls
Documents Downloads Music Pictures
$ ls Downloads
download1.zip
download2.zip
download3.zip
```

```
$ pwd
/Users/me/Documents
$ ls -l
-rwxr-xr-x me staff 21 Sep 06:45 assignment-1.txt
-rwxr-xr-x me staff 09 Feb 03:27 assignment-2.txt
-rwxr-xr-x me staff  1 Jan 01:23 notes-1.txt
-rwxr-xr-x me staff  3 Aug 10:03 notes-2.txt
-rwxr-xr-x me staff  7 Nov 16:21 notes-3.txt
-rwxr-xr-x me staff 27 Sep 04:56 notes-4.txt
```

Where am I?

pwd (print working directory) – get current working directory.

```
$ pwd
/Users/me
```

Navigating your directories –

cd (change directory) – change directory

```
$ pwd
/Users/me
$ ls
Documents Downloads Music Pictures
$ cd Documents
$ pwd
/Users/me/Documents
```

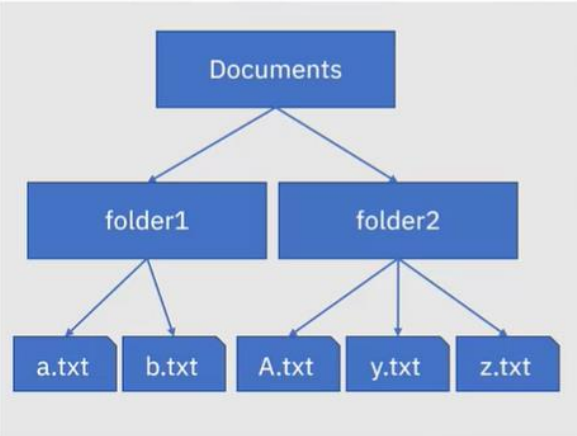
Relative and absolute navigation –

cd (change directory) – change directory

```
$ pwd
/Users/me/Documents/Academics/Math/Notes
$ cd ..
$ pwd
/Users/me/Documents/Academics/Math
$ cd ~
$ pwd
/Users/me
$ cd /Users/me/Documents/Academics/Math/Notes
$ pwd
/Users/me/Documents/Academics/Math/Notes
```

Finding files –

find – find files in directory tree




```
$ pwd
/Users/me/Documents
$ find . -name "a.txt"
./folder1/a.txt
$ find . -iname "a.txt"
./folder1/a.txt
./folder2/A.txt
```

Summary: -

- ***ls*** lists files and directories in a directory.
- ***cd*** allows you to navigate directories.

4] File and Directory Management Commands

Creating directories –

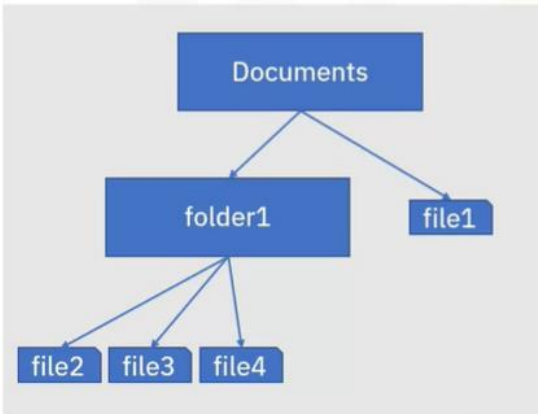
mkdir (make directory) – make directory

```
$ pwd
/Users/me/Documents
$ ls

$ mkdir test
$ ls
test
```

Removing files and directories –

rm (remove) – Remove file or directory




```
$ pwd
/Users/me/Documents
$ ls
file1 folder1
$ rm file1
$ ls
folder1
$ rm folder1
rm: folder1: is a directory
$ rm -r folder1
$ ls

$ mkdir empty_folder
$ rmdir empty_folder
```

Creating files –

touch – Create empty file, update file date

```
$ pwd
/Users/me/Documents
$ touch a.txt b.txt c.txt d.txt
$ ls
a.txt b.txt c.txt d.txt
$ date -r notes.txt
Mon  8 Nov 2021 16:37:45 EST
$ touch notes.txt
$ date -r notes.txt
Fri 12 Nov 2021 10:46:03 EST
```

Copying files and directories –

cp (copy) – Copy files or directory to destination.

To copy files:

```
$ cp /source/file /dest/filename
$ cp /source/file /dest/
```

To copy directories:

```
$ cp -r /source/dir/ /dest/dir/
```

```
$ ls
notes.txt Documents
$ cp notes.txt Documents
$ ls Documents
notes.txt
$ cp -r Documents Docs_copy
$ ls
notes.txt Documents Docs_copy
$ ls Docs_copy
notes.txt
```

Moving files and directories –

mv (move) – Move a file or directory

To copy files:

```
$ mv /source/file /dest/dir/
```

To move directories:

```
$ mv /source/dir/ /dest/dir/
```

```
$ ls
my_script.sh Scripts Notes Documents
$ mv my_script.sh Scripts
$ ls my_script.sh

$ ls Scripts
my_script.sh
$ mv Notes Scripts Documents
$ ls
Documents
$ ls Documents
Scripts Notes
```

Managing files permissions –

chmod (change mode) – Change file permissions.

```
$ ls -l my_script.sh
-rw-r--r- my_script.sh
$ ./my_script.sh
bash: permission denied: ./my_script.sh
$ chmod +x my_script.sh
$ ls -l my_script.sh
-rwxr-xr-x my_script.sh
$ ./my_script.sh
Learning Linux is fun!
```

Summary: -

- ***touch*** allows you to create a file or update its last-modified timestamp.
- ***mkdir*** creates directories and ***rmdir*** deletes empty directories.
- ***cp*** copies files and directories and ***mv*** moves them.
- ***chmod*** modifies file permissions.

Reading: Security - Managing File Permissions and Ownership

Why do we need file permissions and ownership?

Linux is a multi-user operating system. This means that by default, other users will be able to view any files you store on the system. However, you may have some files - such as your personal tax documents or your employer's intellectual property documents - that are private or confidential. How can you protect these sensitive documents from being viewed or modified by others?

File ownership and permissions

There are three possible levels of file ownership in Linux: user, group, and other. Whoever creates a file, namely the **user** at the time of creation, becomes the owner of that file by default. A **group** of users can also share ownership of a file. The **other** category essentially refers anyone in the universe with access to your Linux machine - careful when assigning ownership permission to this level!
Only an official owner of a file is allowed to change its permissions. This means that only owners can decide who can read the file, write to it, or execute it.

Viewing file permissions

Let's say you've entered the following lines of code:

```
$ echo "Who can read this file?" > my_new_file
$ more my_new_file
Who can read this file?
$ ls -l my_new_file
-rw-r--r-- 1 theia users 25 Dec 22 17:47 x
```

Here we've echoed the string `"Who can read this file?"` into a new file called `my_new_file`. The next line uses the `more` command to print the contents of the new file. Finally, the `ls` command with the `-l` option displays the file's (default) permissions: `rw-r--r--`

The first three characters (`rw-`) define the **user** permissions, the next three (`r--`) the **group** pemissions, and the final three (`r--`) the **other** permissions.

So you, being the user, have the permission `rw-`, which means you have read and write permissions by default, but do not have execution permissions. Otherwise there would be an `x` in place of the last `-`.

Thus by looking at the entire line, `rw-r--r--`, you can see that anyone can read the file, nobody can execute it, and you are the only user that can write to it.

Note: *The `-` at the very beginning of the line in the terminal means that the permissions are referring to a file. If you were getting the permissions to a directory, you would see a `d` in the front for "directory".*

Directory permissions

The permissions for directories are similar but distinct for files. Though directories use the same `rwx` format, the symbols have slightly different meanings.

The following table illustrates the meanings of each permission for directories:

Directory Permission	Permissible action(s)
`r`	List directory contents using `ls` command
`w`	Add or remove files or directories
`x`	Enter directory using `cd` command

Setting appropriate permissions on directories is a best practice for both security and stability reasons. Though this reading focuses on security, you will learn more about other reasons for setting file permissions and ownership later in this course.

Making a file private

You can revoke read permissions from your group and all other users by using the `chmod` command. Ensure successful modification by using the `ls -l` command again:

```
1. chmod go-r my_new_file
2. ls -l my_new_file
3. -rw------- 1 theia users 24 Dec 22 18:49 my_new_file
```

In the `chmod` command, `go-r` is the permission change to be applied, which in this case means removing for the group (`g`) and others (`o`) the read (`r`) permission. The `chmod` command can be used with both files and directories.

Executable files - looking ahead

You've learned what it means to read or write to a file, but what does it mean to have permissions to **execute** a file in Linux?
A Linux file is executable if it contains instructions that can be directly interpreted by the operating system. Basically, an exectuable file is a ready-to-run program. They're also referred to as **binaries** or **executables**.
In this course, you will become very familiar with a particular kind of executable called a **script**, which is a program written in a scripting language. You'll learn all about **shell scripting**, or more specifically **Bash scripting**, which is writing scripts in Bash (*born-again shell*), a very popular shell scripting language. A shell script is a plain text file that can be interpreted by a shell.
Formally speaking, for a text file to be considered an executable shell script for a given user, it needs to have two things:

1. Execute permissions set for that user
2. A directive, called a "shebang", in its first line to declare itself to the operating system as a binary

All of this will become more clear to you soon when we get to the topic of shell scripting.

Summary

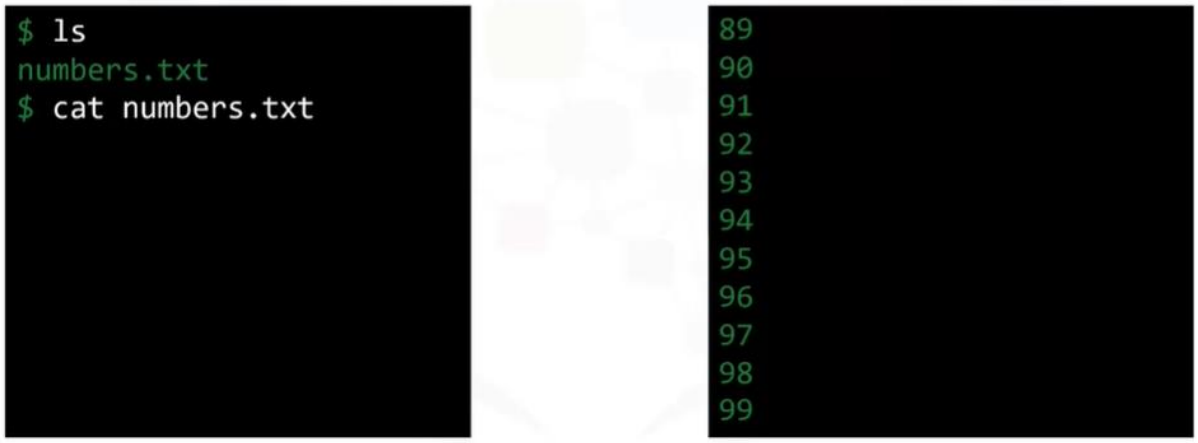
In this reading, you learned that:

- There are three possible levels of file ownership in Linux - user, group, and other - which determine who can read, write to, and execute a file
- You can use the `ls -l` command to view file and directory permissions
- You can change permissions on a file by using the `chmod` command

5] Viewing File Contents

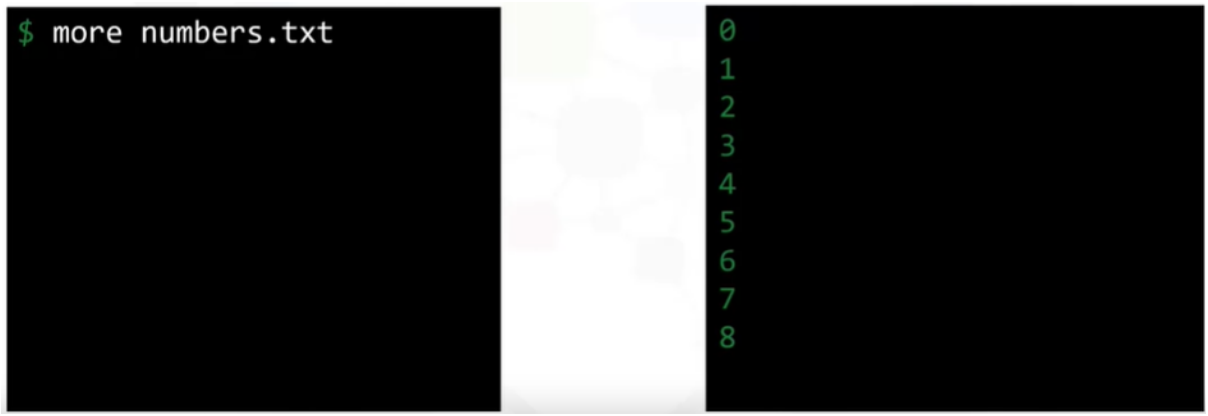
Viewing your file all at once –

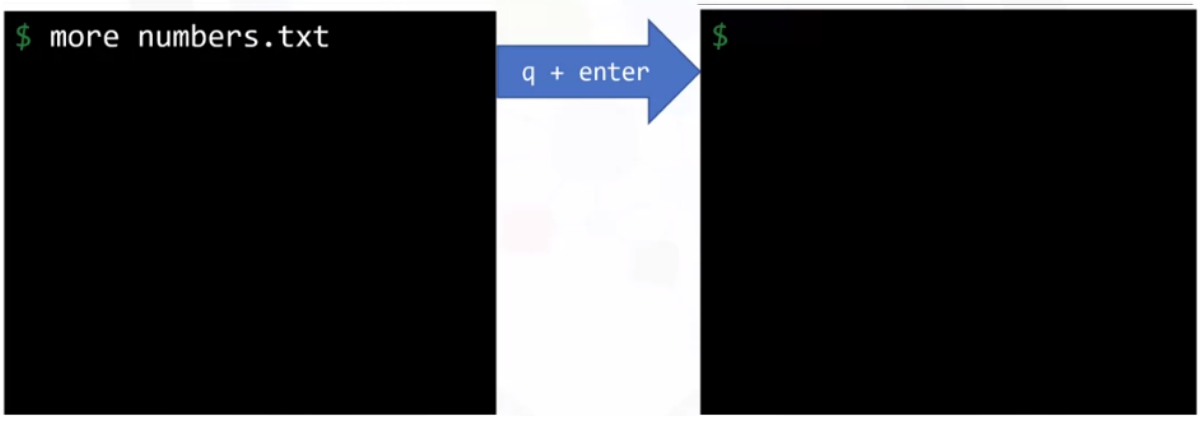
cat (“catenate”) – Print entire file contents



Viewing your file page-by-page –

more – Print file contents page-by-page





- Entering "q" quits the “more program” and returns you to the command prompt.

Viewing the first 10 lines –

head – Print first 10 lines of file

```
$ head numbers.txt
0
1
2
3
4
5
6
7
8
9
```

head – Print first N lines of file

```
$ head -n 3 numbers.txt
0
1
2
```

Viewing the last 10 lines –

tail – Print last 10 lines of file

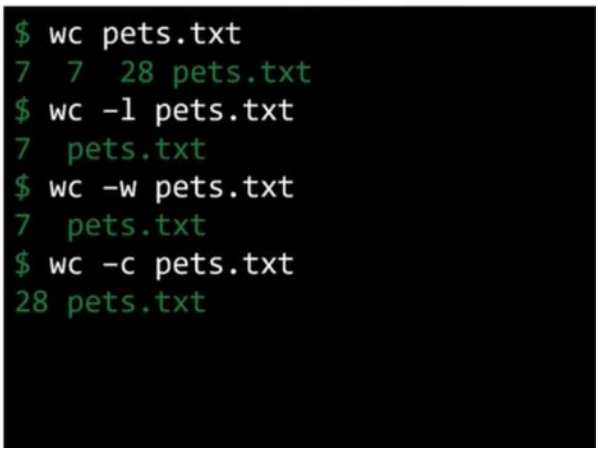
```
$ tail numbers.txt
90
91
92
93
94
95
96
97
98
99
```

tail – Print last N lines of file

```
$ tail -n 3 numbers.txt
97
98
99
```

Counting lines, words, and characters –

wc (word count) – Count characters, words, lines



To view only line count → wc -l

To view only word count → wc -w

To view only character count →wc -c

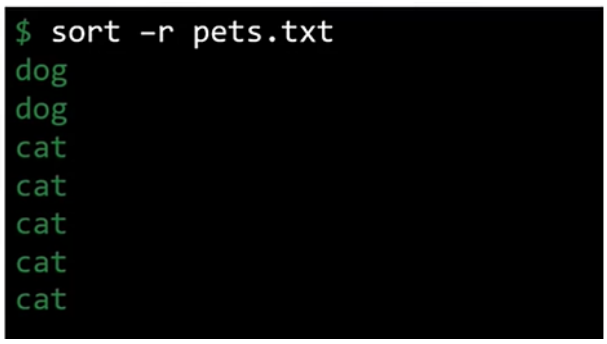
Summary: -

- Apply the ***cat***, ***more***, ***head***, and ***tail*** commands to view file contents in multiple ways.
- Determine the line, word, and character count of a file using ***wc***.

6] Customizing View of File Content

Sorting your views line-by-line –

sort – Sort lines in a file



Excluding repetitive lines from views –

uniq (“unique”) – Filter out repeated lines

```
$ cat pets.txt
cat
cat
cat
cat
dog
dog
cat
```

```
$ uniq pets.txt
cat
dog
cat
```

Extracting lines matching a pattern –

grep (“global regular expression print”) – Return lines in file matching pattern

```
$ cat people.txt
Alan Turing
Bjarne Stroustrup
Charles Babbage
Dennis Ritchie
Erwin Schrodinger
Fred Hoyle
Guido Rossum
Henri Poincare
Ivan Pavlov
John Neumann
Ken Thompson
```

```
$ grep ch people.txt
Dennis Ritchie
Erwin Schrodinger

$ grep -i ch people.txt
Charles Babbage
Dennis Ritchie
Erwin Schrodinger
```

- -i option expands the pattern search by making it case-insensitive

Extracting slices from lines –

cut – Extracts a section from each line

```
$ cat people.txt
Alan Turing
Bjarne Stroustrup
Charles Babbage
Dennis Ritchie
Erwin Schrodinger
Fred Hoyle
Guido Rossum
Henri Poincare
Ivan Pavlov
John Neumann
Ken Thompson
```

```
$ cut -c 2-9 people.txt
lan Turi
jarne St
harles B
ennis Ri
rwin Sch
red Hoyl
uido Ros
enri Poi
van Pavl
ohn Neum
en Thomp
```

Extracting fields from lines –

cut – Extracts a field from each line

```
$ cat people.txt
Alan Turing
Bjarne Stroustrup
Charles Babbage
Dennis Ritchie
Erwin Schrodinger
Fred Hoyle
Guido Rossum
Henri Poincare
Ivan Pavlov
John Neumann
Ken Thompson
```

```
$ cut -d ' ' -f2 people.txt
Turing
Stroustrup
Babbage
Ritchie
Schrodinger
Hoyle
Rossum
Poincare
Pavlov
Neumann
Thompson
```

-f2 → To specify 2nd field from each line

-d ‘ ‘ → To specify field delimiter as space.

Merging lines from multiple files –

paste – Merge lines from different files

```
$ cat first.txt
Alan
Bjarne
Charles
Dennis
Erwin
Fred
Guido
Henri
Ivan
John
Ken
```

```
$ cat last.txt
Turing
Stroustrup
Babbage
Ritchie
Schrodinger
Hoyle
Rossum
Poincare
Pavlov
Neumann
Thompson
```

```
$ cat yob.txt
1912
1950
1791
1941
1887
1915
1956
1854
1849
1903
1943
```

```
$ paste first.txt last.txt yob.txt
Alan    Turing      1912
Bjarne  Stroustrup  1950
Charles Babbage     1791
Dennis  Ritchie     1941
Erwin   Schrodinger 1887
Fred    Hoyle       1915
Guido   Rossum      1956
Henri   Poincare    1854
Ivan    Pavlov      1849
John    Neumann     1903
Ken     Thompson    1943
```

```
$ paste -d "," first.txt last.txt yob.txt
Alan,Turing,1912
Bjarne,Stroustrup,1950
Charles,Babbage,1791
Dennis,Ritchie,1941
Erwin,Schrodinger,1887
Fred,Hoyle,1915
Guido,Rossum,1956
Henri,Poincare,1854
Ivan,Pavlov,1849
John,Neumann,1903
Ken,Thompson,1943
```

-d " , " → To specify field delimiter as comma.

Summary: -

- Use ***grep*** to extract lines containing a pattern
- Use ***sort***, ***unique***, ***cut*** and ***paste*** to create views from file contents.

7] File Archiving and Compression Commands

Archiving and compression –

Archives:

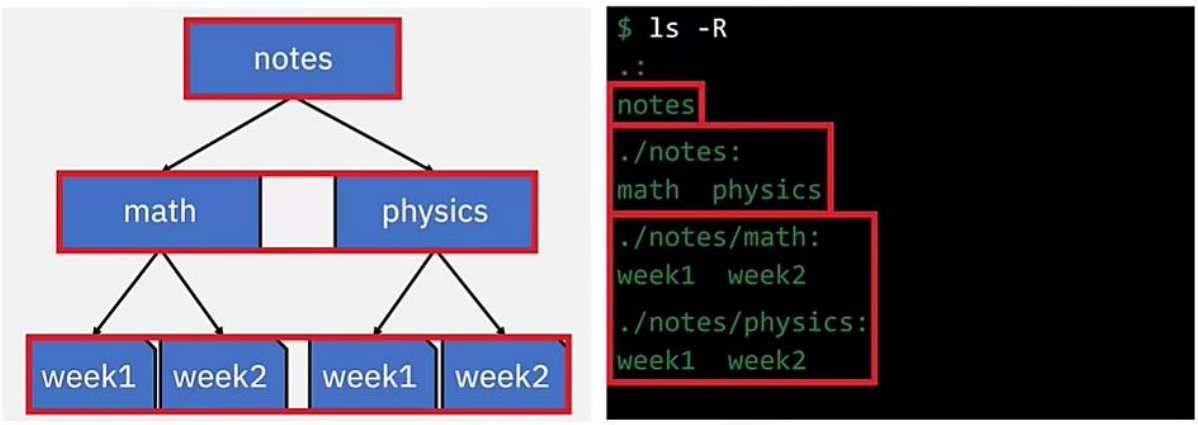
- Store rarely used information and preserve it
- Are a collection of data files and directories stored as a single file
- Make the collection more portable and serve as a backup in case of loss or corruption

File Compression:

- Reduces file size by reducing information
- Preserves storage space, speeds up data transfer, and reduces bandwidth load

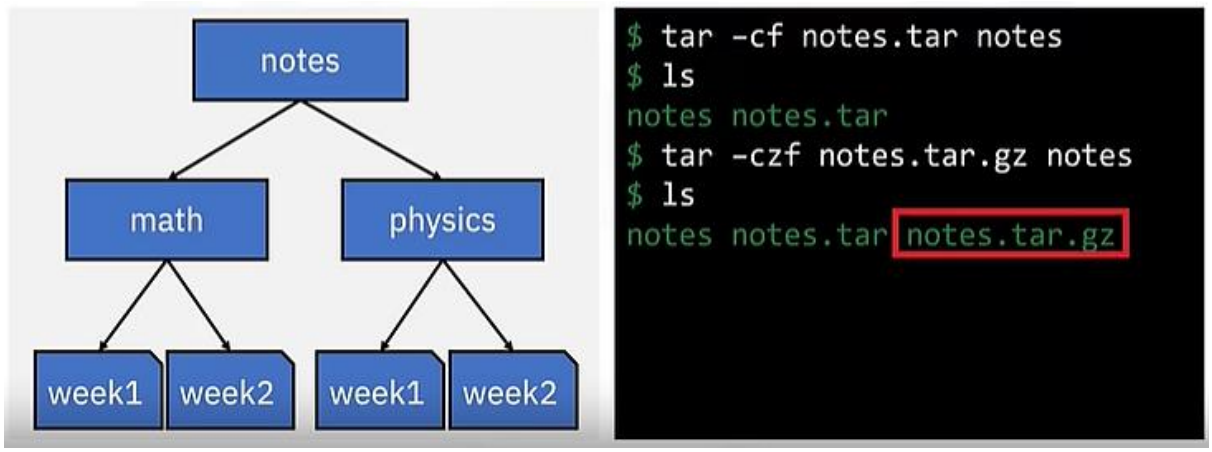
Directory tree archiving –

Notes directory tree example



File archiving and compression –

tar (tape archiver) – Archive and extract files

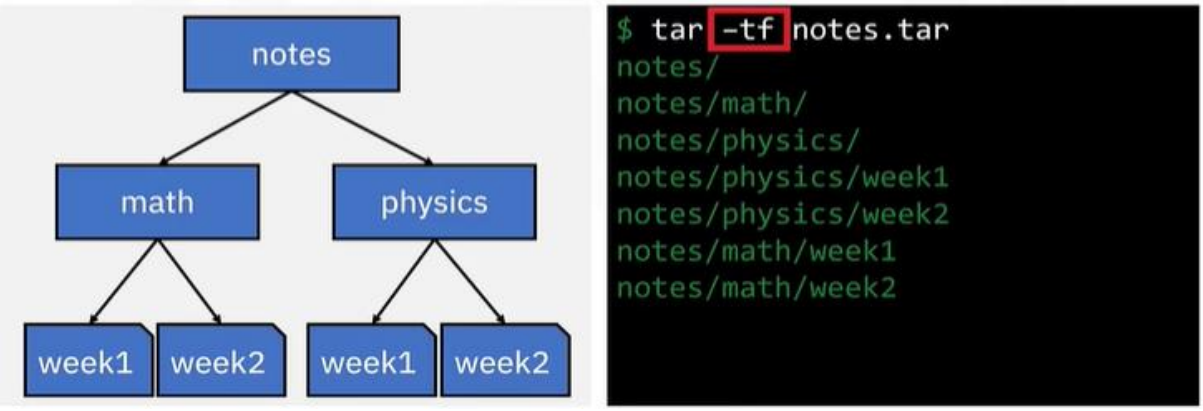


-cf → a name for the archived file followed by followed by the file or directory you wish to archive in which c means create a new archive and f tells tar to interpret its input from file rather than from the default, which is standard input.

-czf → which filters the archive file through a GNU (pr. “geh-noo”) compression program called g-zip.

Checking your archive contents –

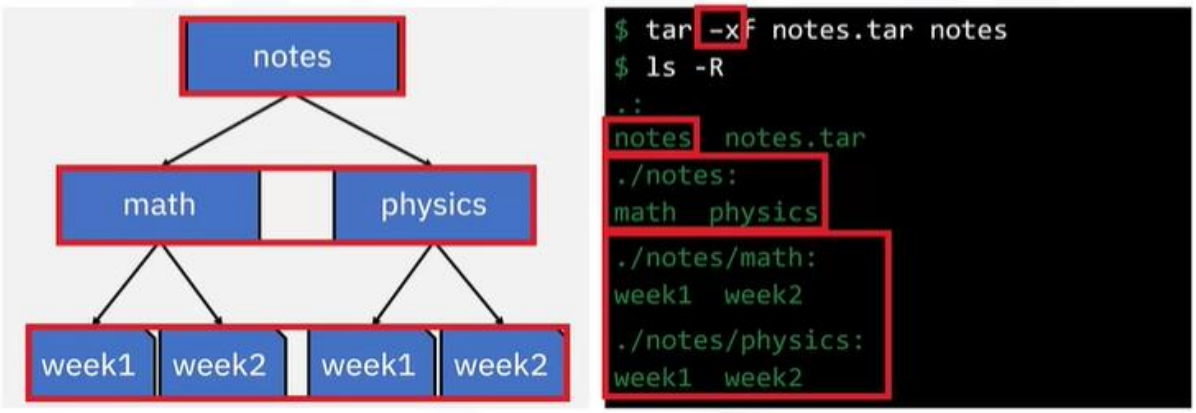
tar – List archive contents



-t → lists all the files and directories in your tar ball.

Extracting archived files –

tar – Extract files and folders

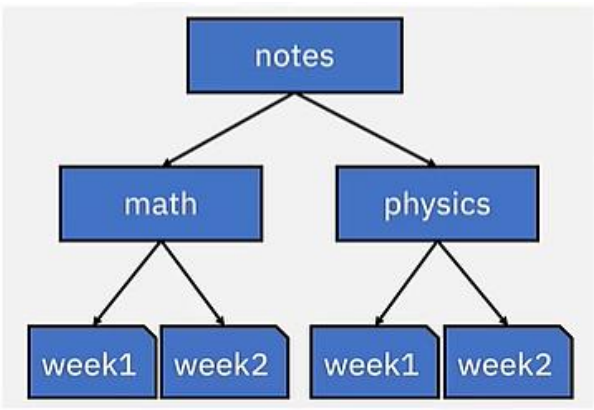


-x → tells tar to extract file and directory objects from the archive.

ls -R → the archived notes folder has been de-archived into a parent folder called notes

Decompressing and extracting archives –

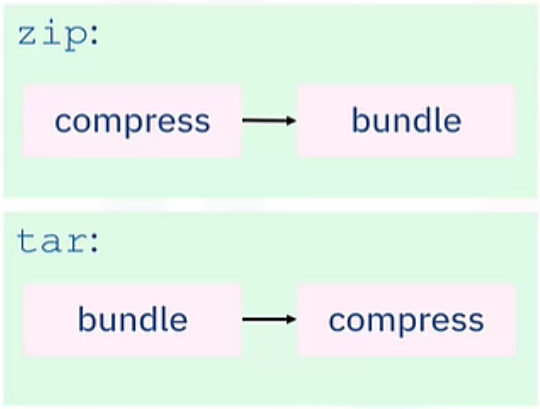
tar – Decompress and extract




```
$ tar -xzf notes.tar.gz notes
$ ls -R
.:
notes  notes.tar
./notes:
math  physics
./notes/math:
week1  week2
./notes/physics:
week1  week2
```

File compression and archiving –

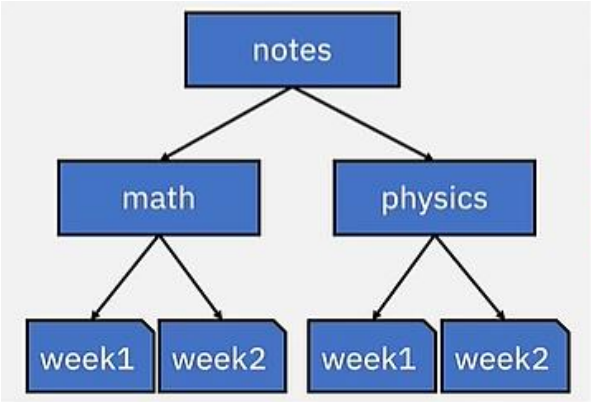
zip – Compress files and directories to an archive




```
$ zip notes.zip notes
$ ls
notes  notes.zip
```

Extracting and decompressing archives –

unzip – Extract and decompress zipped archive




```
$ unzip notes.zip
$ ls -R
.:
notes  notes.tar
./notes:
math  physics
./notes/math:
week1  week2
./notes/physics:
week1  week2
```

Summary: -

- Compression preserves storage space, speeds data transfer, and reduces system load
- ***zip*** compresses files and folders prior to archiving them
- ***tar*** archives files and directories into a tarball, and can also compress it.
- ***unzip*** unpacks and decompress a zipped archive
- ***tar*** decompresses and unpacks a ***tar.gz*** archive

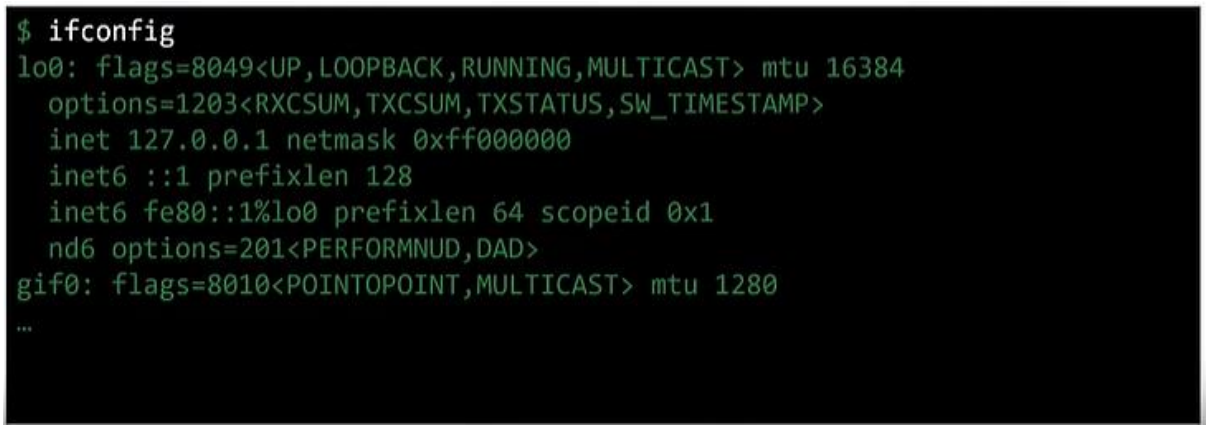
8] Networking Commands

Getting your machine's host name –

hostname – Print host name

Getting network information –

ifconfig (Interface configuration) – Display or configure the system network interfaces



Getting ethernet adapter info –

```
$ ifconfig eth0
eth0: Link encap:Ethernet  HWaddr 00:0B:CD:1C:18:5A
  inet addr:172.16.25.126  Bcast:172.16.25.63  Mask:255.255.255.224
  inet6 addr: fe80::20b:cdff:fe1c:185a/64 Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
  RX packets:2345583 errors:0 dropped:0 overruns:0 frame:0
  TX packets:2221421 errors:0 dropped:0 overruns:0 carrier:0
  collisions:0 txqueuelen:1000
  RX bytes:293912265 (280.2 MiB)  TX bytes:1044100408 (995.7 MiB)
  Interrupt:185 Memory:f7fe0000-f7ff0000
```

Testing server connections –

ping – Send ICMP packets to URL and print response

```
$ ping www.google.com
PING www.google.com (142.251.41.68): 56 data bytes
64 bytes from 142.251.41.68: icmp_seq=0 ttl=119 time=21.750 ms
64 bytes from 142.251.41.68: icmp_seq=1 ttl=119 time=20.712 ms
64 bytes from 142.251.41.68: icmp_seq=2 ttl=119 time=24.065 ms
64 bytes from 142.251.41.68: icmp_seq=3 ttl=119 time=36.751 ms
64 bytes from 142.251.41.68: icmp_seq=4 ttl=119 time=41.774 ms
^C
--- www.google.com ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 20.712/229.010/41.774/9.603 ms
```

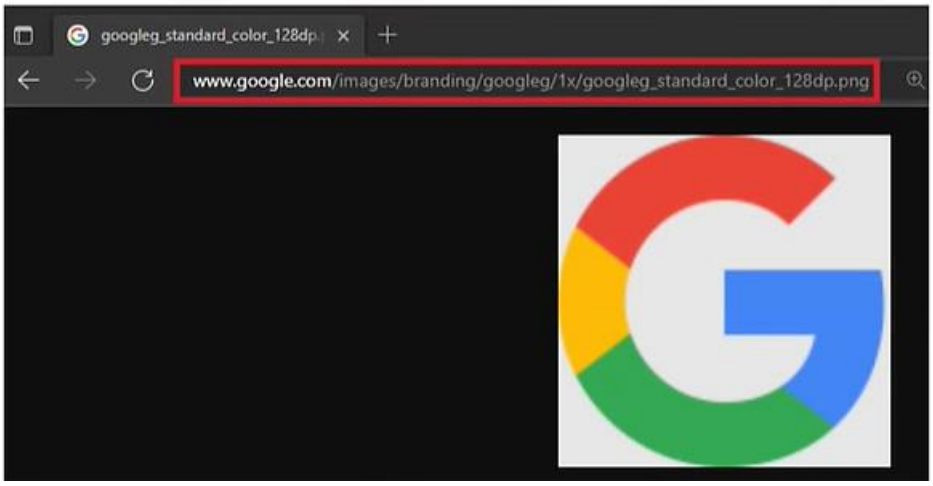
```
$ ping -c 5 www.google.com
PING www.google.com (142.251.41.68): 56 data bytes
64 bytes from 142.251.41.68: icmp_seq=0 ttl=119 time=17.491 ms
64 bytes from 142.251.41.68: icmp_seq=1 ttl=119 time=19.784 ms
64 bytes from 142.251.41.68: icmp_seq=2 ttl=119 time=24.279 ms
64 bytes from 142.251.41.68: icmp_seq=3 ttl=119 time=24.964 ms
64 bytes from 142.251.41.68: icmp_seq=4 ttl=119 time=26.106 ms

--- www.google.com ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 17.491/22.525/26.106/3.308 ms
```

Web scraping with curl –

curl (Client URL) – Transfer data to and from URL(s)

```
$ curl www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
lang="en-CA"><head><meta content="text/html; charset=UTF-8" http-
equiv="Content-Type"><meta
content="/images/branding/googleg/1x/googleg_standard_color_128dp.png"
itemprop="image"><title>Google</title><script
nonce="gPa6M7RHuxLHFwYnP5CH4A==">(function(){window.google={kEI:'FdCKYdj-
LrOt0PEPuoqLIA',kEXPI:'0,18168,1284368,56873,1709,4350,206,4804,2316,383,
246,5,1354,5250,1122516,1197719,329548,51224,16114,17444,11240,17572,4859
,1361,9291,3027,2816,1931,12834,4020,978,13228,516,3331,4192,6430,7432,14
390,919,5081,887,706,1279,2212,530,149,1103,840,1983,213,4101,3514,606,20
...
```



Scraping a web page's HTML to file –

curl (Client URL) – Transfer data to and from URL(s)

```
$ curl www.google.com -o google.txt
$ head -n 1 google.txt
<!doctype html><html itemscope=""
itemtype="http://schema.org/WebPage" lang="en-CA"><head><meta
content="text/html; charset=UTF-8" http-equiv="Content-
Type"><meta content="/images/branding/googleg/1x/googleg_standard_color_1
28dp.png" itemprop="image"><title>Google</title><script nonce="gPa6M7RHux
LHFwYnP5CH4A==">(function(){window.google={kEI:'FdCKYdj-
LrOt0PEPuoqLIA',kEXPI:'0,18168,1284368,56873,1709,4350,206,4804,2316,383,
246,5,1354,5250,1122516,1197719,329548,51224,16114,17444,11240,17572,4859
...
```

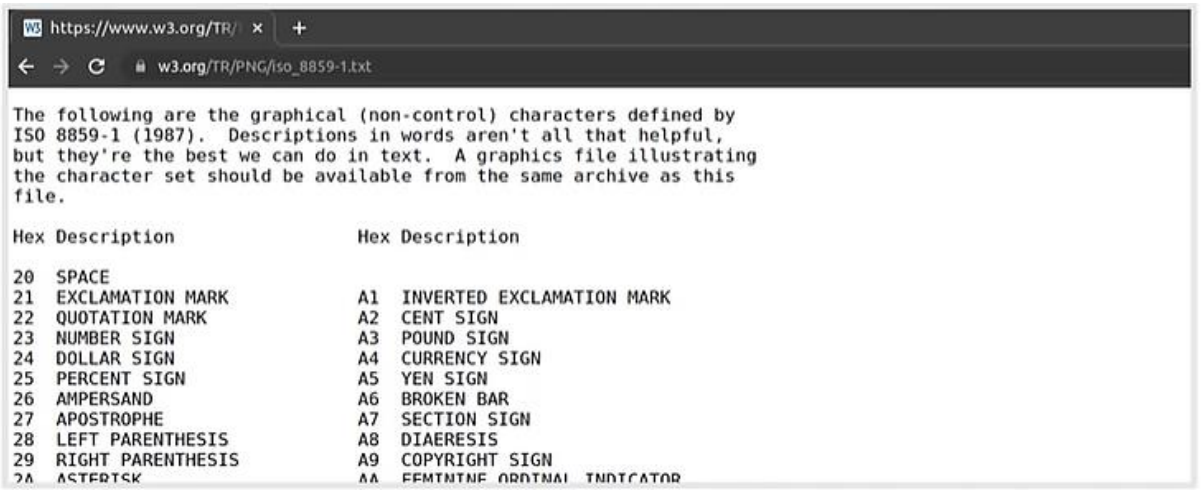
Downloading files from a URL –

wget (Web get) – Download files from a URL

- more focused than curl, supports recursive file downloads

```
$ wget https://www.w3.org/TR/PNG/iso_8859-1.txt
Resolving www.w3.org (www.w3.org)... 128.30.52.100
Connecting to www.w3.org (www.w3.org)|128.30.52.100|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 6121 (6.0K) [text/plain]
Saving to: 'iso_8859-1.txt'

iso_8859-
1.txt                       100%[===================================>]
  5.98K  --.-KB/s    in 0s
```




```
$ head -n 12 iso_8859-1.txt
The following are the graphical (non-control) characters defined by
ISO 8859-1 (1987).  Descriptions in words aren't all that helpful,
but they're the best we can do in text.  A graphics file illustrating
the character set should be available from the same archive as this
file.

Hex Description                Hex Description

20  SPACE
21  EXCLAMATION MARK           A1  INVERTED EXCLAMATION MARK
22  QUOTATION MARK             A2  CENT SIGN
23  NUMBER SIGN                A3  POUND SIGN
```

Summary: -

- View network configuration with ***hostname*** and ***ifconfig***.
- Test a network connection using the ***ping*** command.
- Send and receive data using the ***curl*** and ***wget*** commands.

Reading (Optional): A Brief Introduction to Networking

Computer Networks

A **computer network** is a set of computers that are able to communicate with each other and share **resources** provided by **network nodes**.
Examples of computer networks include Local Area Networks (LANs), Wide Area Networks (WANs), and the entire Internet. The Internet, or World Wide Web, is essentially a giant network of computer networks.

A network **resource** is any object, such as a file or document, which can be *identified* by the network.
An object is identifiable if it can be assigned a unique name and address that the network can use to identify and access it.

A **network node** is any device that participates in a network.
A network can include any device which is not necessarily a computer but is part of the network's infrastructure. Examples of network nodes include modems, network switches, hubs, and wifi hotspots.

Hosts, Clients, and Servers

A **host** is a special type of node in a computer network - it is a computer that can function as a **server** or a **client** on a network.
A **server** is a host computer that is able to accept a connection from a **client** host and fulfill certain resource requests made by the client.
Many hosts can perform either role, acting as both client and server.

Packets and Pings

A **network packet** is a formatted chunk of data that can be transmitted over a network.
Today's computer networks typically use communication protocols that are based on such packets of information. Every packet consists of two types of data: 1. the **control information**, and 2. the **payload**. The control information is data about how and where to deliver the payload, such as the source and destination network addresses, while the payload is the message being sent.
The `ping` command works by sending special 'echo request' packets to a host and waiting for a response from the host.
`ping` is a utility available on most operating systems that have networking capability. Linux has its own implementation of the `ping` command that's used to test and troubleshoot connectivity to other network hosts.

URLs and IP Adresses

IP stands for "Internet Protocol" which defines the format of data transmitted over the internet or a local network.

An **IP address** is a code used to uniquely identify any host on a network.
An IP address can be used to establish a connection to a host and exchange packets with it, for example using the `ping` command. In addition to their payload, IP packets - a type of network packet that conforms to the Internet Protocol - contain the IP addresses of the source and destination hosts.

A **URL**, more commonly known as a web address, stands for *Uniform Resource Locator*.
A URL uniquely identifies a web resource and enables access to that resource. Typically the resource that a URL points to is a web page, but it can also be used for tasks such as transferring files, sending emails, and accessing databases.

For example, the URL to the Wikipedia page for URL is `https://en.wikipedia.org/wiki/URL`. Just like for a typical URL, its format indicates a protocol (`https`), a hostname (`en.wikipedia.org`), and a file name (`/wiki/URL`).

Summary

In this reading, you learned that:

- A computer network is a set of computers that are able to communicate with each other and share resources. A network resource is any object, such as a file or document, which can be identified by the network. A network node is any device that participates in a network.
- A host is a computer that can function as a server or a client on a network. A server is a host computer that is able to accept a connection from a client host and fulfill certain resource requests made by the client.
- A network packet is a formatted chunk of data that can be transmitted over a network. The `ping` command works by sending special 'echo request' packets to a host and waiting for a response from the host.
- An IP address is a code used to uniquely identify any host on a network. A URL identifies a web resource and enables access to that resource.

Summary & Highlights:-

- A shell is an interactive user interface
- You can use shell commands for navigating and working with files and directories, and to zip and unzip files
- You can use the “curl” and “wget” commands to display and download files from URLs
- The “echo” command prints string or variable values
- The “cat” and “tail” commands display file contents
- You can get user information with the "whoami" and "id" commands
- You can check system disk usage using the "df" command
- The “ls” command lists all files and directories contained within a specified directory tree
- The “cd” command allows you to navigate directories
- The “touch” command allows you to create a file or update its last–modified timestamp
- The “mkdir” command creates directories and “rmdir” deletes empty directories
- You can determine line, word, and character counts with “wc”
- You can use “grep” to get the lines of a file matching your desired criteria
- The “tar” command decompresses and unpacks a “tar.gz” archive
- You can view network configuration with “hostname” and “ifconfig”

Week 3

Introduction to Shell Scripting

Module 1: - Introduction to Shell Scripting

1] Shell Scripting Basics

What is a script?

- Script: list of commands interpreted by a scripting language.
- Commands can be entered interactively or listed in a text file.
- Scripting languages are interpreted at runtime.
- Scripting is slower to run, but faster to develop.

What is a script used for?

- Widely used to automate processes.
- ETL jobs, file backups and archiving, system admin.
- Used for application integration, plug-in development, web apps, and many other tasks.

Shell scripts and the 'shebang' –

- Shell script – executable text file with an *interpreter directive*
- Aka 'shebang' directive:

  #!interpreter [optional-arg]

  'interpreter' – path to an executable program
  'optional-arg' – single argument string.

Example – 'shebang' directives

- **Shell script directives:**

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

- **Python script directive:**

```
#!/usr/bin/env python3
```

'Hello World' example shell script –

Create the shell script:

```
$ touch hello_world.sh
$ echo '#! /bin/bash' >> hello_world.sh
$ echo 'echo hello world' >> hello_world.sh
```

Make it executable:

```
$ ls -l hello_world.sh
-rw-rw-r-- 1 jgrom jgrom 12 Jun 27 09:13 hello_world.sh
$ chmod +x hello_world.sh
$ ls -l hello_world.sh
-rwxrwxr-x 1 jgrom jgrom 12 Jun 27 09:13 hello_world.sh
```

Run your bash script:

```
$ ./hello_world.sh
hello world
```

Summary: -

- A shell script is a program that begins with a 'shebang' directive.
- Shell scripts are used to run commands and programs.
- Scripting languages are interpreted, not complied.
- Compiled languages are faster, but take more time to develop.

Reading: A Brief Introduction to Shell Variables

What is a shell variable?

Shell variables offer a powerful way to store and later access or modify information such as numbers, character strings, and other data structures by name. Let's look at some basic examples to get the idea.

Consider the following example.

```
$ firstname=Jeff
$ echo $firstname
Jeff
```

The first line assigns the value `Jeff` to a new variable called `firstname`. The next line accesses and displays the value of the variable, using the `echo` command along with the special character `$` in front of the variable name to extract its value, which is the string `Jeff`.
Thus, we have created a new shell variable called `firstname` for which the value is `Jeff`.
This is the most basic way to create a shell variable and assign it to a value all in one step.

Reading user input into a shell variable at the command line

Here's another way to create a shell variable, using the `read` command. After entering

```
$ read lastname
```

on the command line, the shell waits for you to enter some text:

```
$ read lastname
Grossman
$
```

Now we can see that the value `Grossman` has just been stored in the variable `lastname` by the `read` command:

```
$ read lastname
Grossman
$ echo $lastname
Grossman
```

By the way, notice that you can echo the values of multiple variables at once.

```
1. $ echo $firstname $lastname
2. Jeff Grossman
```

As you will soon see, the `read` command is particularly useful in shell scripting. You can use it within a shell script to prompt users to input information, which is then stored in a shell variable and available for use by the shell script while it is running. You will also learn about **command line arguments**, which are values that can be passed to a script and automatically assigned to shell variables.

Summary

In this reading, you learned that:

- Shell variables store values and allow users to later access them by name
- You can create shell variables by declaring a shell variable and value or by using the `read` command

2] Filters, Pipes, and Variables

Pipes and filters –

Filters are shell commands, which:

- Take input from standard input
- Send output to standard output
- Transform input data into output data
- Examples are wc, cat, more, head, sort, …
- Filters can be chained together

Pipe command – |

- For chaining filter commands

```
command1 | command2
```

- Output of command 1 is input of command 2
- Pipe stands for pipeline.

Shell variables –

- Scope limited to shell
- Set – list all shell variables

```
$ set | head -4
BASH=/usr/bin/bash
BASHOPTS=checkwinsize:cm
  dhist:complete_fullquot
  e:expand_aliases:extglo
  b:extquote:force_fignor
  e:globasciiranges:hista
  ppend:interactive_comme
  nts:progcomp:promptvars
  :sourcepath
BASH_ALIASES=()
BASH_ARGC=([0]="0")
```

Defining shell variables –

```
var_name=value
```

- No spaces around '='

```
$ GREETINGS="Hello"
$ echo $GREETINGS
Hello

$ AUDIENCE='World'
$ echo $GREETINGS $AUDIENCE
Hello World

$ unset AUDIENCE
```

```
unset var_name
```

- deletes var_name

Environment variables –

- Extended scope

```
export var_name
```

```
$ export GREETINGS


$ env | grep "GREE"
$ GREETINGS=Hello
```

- env – list all environment variables.

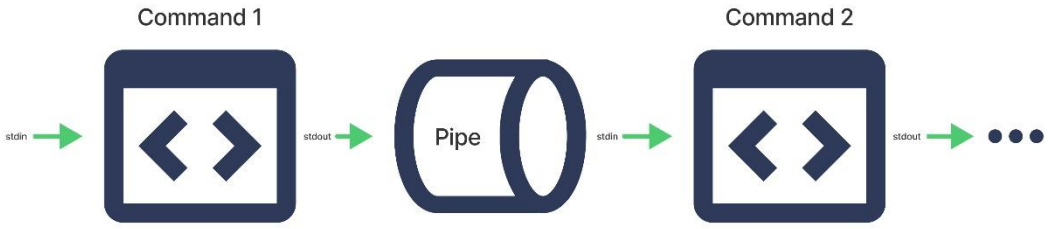
Summary: -

- Filters are shell commands.
- The pipe operator allows you to chain filter commands.
- Shell variables can be assigned values with '=' and listed using 'set'.
- Environment variables are shell variables with extended scope; create with 'export,' list with 'env'.

Reading: Examples of Pipes

What are pipes?

Put simply, pipes are commands in Linux which allow you to use the output of one command as the input of another.



Pipes "|" use the following format:

`[command 1] | [command 2] | [command 3] ... | [command n]`
(no limit to the number of times in a row u can pipe!)

Pipe examples

Ex 1: Combining commands

Let's start with a commonly used example.

Recall the commands:

- sort: sorts the lines of text in a file and displays the result
- uniq: prints input with consecutive repeated lines collapsed to a single, unique line

With the help of the pipe operator, you can combine these commands to print all the unique lines in a file!

Suppose you have the file pets.txt with the following contents:

```
$ cat pets.txt
goldfish
dog
cat
```

```
parrot
dog
goldfish
goldfish
```

If you *only* use sort on pets.txt, you get:

```
$ sort pets.txt
cat
dog
dog
goldfish
goldfish
goldfish
parrot
```

And if you *only* use uniq, you get:

```
$ uniq pets.txt
goldfish
dog
cat
parrot
dog
goldfish
```

This time, you removed consecutive duplicates, but non-consecutive duplicates of "dog" and "goldfish" remain.

But by combining the two commands in the correct order - by first using sort then uniq - you get back:

```
$ sort pets.txt | uniq
cat
dog
goldfish
parrot
```

Since sort sorts all identical items consecutively, and uniq removes all consecutive duplicates, combining the commands prints only the unique lines from pets.txt!

Ex 2: Applying a command to strings and files

Some commands, such as tr, *only* accept "standard input" as input (not strings or filenames):

- tr (translate) - replaces characters in input text.

  Syntax: tr [OPTIONS] [target characters] [replacement characters]

In cases like this, we can use piping to apply the command to strings and file contents.

With strings, you could, for example, use echo in combination with tr to replace all vowels in a string with underscores, as follows:

```
$ echo "Linux and shell scripting are awesome\!" | tr "aeiou" "_"
L_n_x _nd sh_ll scr_pt_ng _r_ _w_s_m_!
```

To perform the complement of the operation from the previous example, that is, to replace all consonants with an underscore, you can use the -c option like this:

```
$ echo "Linux and shell scripting are awesome\!" | tr -c "aeiou" "_"
_i_u__a_____e_____i__i___a_e_a_e_o_e_
```

With files, you could use cat in combination with tr to change all of the text to upper case as follows:

```
$ cat pets.txt | tr "[a-z]" "[A-Z]"
GOLDFISH
DOG
CAT
PARROT
DOG
GOLDFISH
GOLDFISH
```

The possibilities are endless! For example you could add uniq to the above pipeline to only return unique lines in the file, like to:

```
$ sort pets.txt | uniq | tr "[a-z]" "[A-Z]"
CAT
DOG
```

```
4. GOLDFISH
5. PARROT
```

Ex 3: Extracting information from URLs

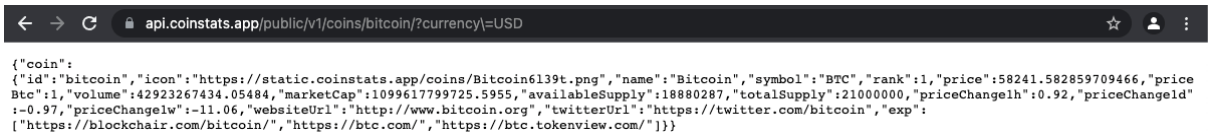
You can even use curl in combination with the grep command to extract components of URL data by piping the output of curl to grep.

Let's see how you can use this pattern to get the current price of BTC (Bitcoin) in USD.

First, you find a public URL API. In this example, you will use one provided by CoinStats.

Specifically, they provide a public API (no key required) https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD which returns some json about the current BTC price in USD.

You can see what this looks like in your browser:



Entering the following command returns the BTC price data, displayed as a json object:

```
1. $ curl -s --location --request GET
   https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD
2. {
3.   "coin": {
4.     "id": "bitcoin",
5.     "icon": "https://static.coinstats.app/coins/Bitcoin6l39t.png",
6.     "name": "Bitcoin",
7.     "symbol": "BTC",
8.     "rank": 1,
9.     "price": 57907.78008618953,
10.    "priceBtc": 1,
11.    "volume": 48430621052.9856,
12.    "marketCap": 1093175428640.1146,
13.    "availableSupply": 18877868,
14.    "totalSupply": 21000000,
```

```
    "priceChange1h": -0.19,
    "priceChange1d": -0.4,
    "priceChange1w": -9.36,
    "websiteUrl": "http://www.bitcoin.org",
    "twitterUrl": "https://twitter.com/bitcoin",
    "exp": [
      "https://blockchair.com/bitcoin/",
      "https://btc.com/",
      "https://btc.tokenview.com/"
    ]
  }
}
```

Note: *For the purpose of this reading, we've reformatted the output to make it easier to interpret. The actual output is a continuous stream of text.*

The json field you want to grab here is "price": [numbers].[numbers]". To grab this you can use the following grep command to extract it from the json text:

grep -oE "\"price\"\s*:\s*[0-9]*?\.[0-9]*"

Let's break down the details of this statement:

- -o tells grep to *only* return the matching portion
- -E tells grep to be able to use extended regex symbols such as ?
- \"price\" matches the string "price"
- \s* matches any number (including 0) of whitespace (\s) characters
- : matches :
- [0-9]* matches any number of digits (from 0 to 9)
- ?\. optionally matches a . (this is in case price were an integer)

Now that you have the grep statement that you need, you can pipe the BTC data to it using the curl command from above:

```
$ curl -s --location --request GET https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD |\
    grep -oE "\"price\":\s*[0-9]*?\.[0-9]*"
"price": 57907.78008618953
```

Tip: The backslash \ character used here after the pipe | allows you to write the expression on multiple lines.

Finally, to get *only* the value in the price field, and drop the "price" label, you can use chaining to pipe the same output to another grep:

```
$ curl -s --location --request GET
https://api.coinstats.app/public/v1/coins/bitcoin\?currency\=USD |\
    grep -oE "\"price\":\s*[0-9]*?\.[0-9]*" |\
    grep -oE "[0-9]*?\.[0-9]*"
57907.78008618953
```

Summary

In this reading, you learned that:

- Pipes are commands in Linux which allow you to use the output of one command as the input of another
- You can combine commands such as sort and uniq to organize strings and text file contents
- You can pipe the output of a curl command to grep to extract components of URL data

3] Useful Features of the Bash Shell

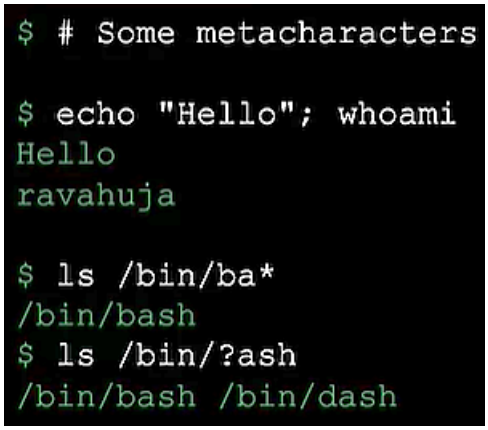
Metacharacters –

- precedes a comment

; - command separator

* - filename expansion wildcard

? – single character wildcard in filename expansion.



Quoting –

\ - escape special character interpretation

“ “ – interpret literally, but evaluate metacharacters

‘ ‘ – interpret literally

```
$ echo "\$1 each"
$1 each

$ echo "$1 each"
 each


$ echo '$1 each'
$1 each
```

I/O redirection –

Input/Output, or I/O redirection, refers to a set of features used for redirecting.
> - Redirect output to file
>> - Append output to file
2> - Redirect standard error to file
2>> - Append standard error to file
< - Redirect file contents to standard input

Examples –

```
$ echo "line1" > eg.txt
$ cat eg.txt
line1
$ echo "line2" >> eg.txt
$ cat eg.txt
line1
line2
```

```
$ garbage
garbage: command
not found
$ garbage 2> err.txt
$ cat err.txt
garbage: command not
found
```

Command substitution –

- Replace command with its output
  $(command) or `command`
- Store output of ‘pwd’ command in ‘here’:

```
$ here=$(pwd)
$ echo $here
/home/jgrom
```

Command line arguments –

- Program arguments specified on the command line

- A way to pass arguments to a shell script
- Usage:

```
$ ./MyBashScript.sh arg1 arg2
```

Batch vs. concurrent modes –
Batch Mode:

- Commands run sequentially

```
command1; command2
```

Concurrent Mode:

- Commands run in parallel

```
command1 & command2
```

Summary: -

- Metacharacters have meaning to the shell.
- Quoting is used to specify whether the shell should interpret special characters as metacharacters, or 'escape' them.
- Input/Output, or I/O redirection, refers to a set of features used for redirecting.
- You can use command substitution when you want to replace a command with its output.
- Command line arguments provide a way to pass arguments to a shell script.
- In concurrent mode, multiple commands can run simultaneously/parallel.

Reading: Examples of Bash Shell Features

Metacharacters

Metacharacters are characters having special meaning that the shell interprets as instructions.

Metacharacter	Meaning
#	Precedes a comment
;	Command separator
*	Filename expansion wildcard
?	Single character wildcard in filename expansion

Pound

The pound # metacharacter is used to represent comments in shell scripts or configuration files. Any text that appears after a # on a line is treated as a comment and is ignored by the shell.

```
#!/bin/bash

# This is a comment
echo "Hello, world!"  # This is another comment
```

Comments are useful for documenting your code or configuration files, providing context, and explaining the purpose of the code to other developers who may read it. It's a best practice to include comments in your code or configuration files wherever necessary to make them more readable and maintainable.

Semicolon ;

The semicolon ; metacharacter is used to separate multiple commands on a single command line. When multiple commands are separated by a semicolon, they are executed sequentially in the order they appear on the command line.

```
$ echo "Hello, "; echo "world!"
Hello,
world!
```

As you can see from the example above, the output of each echo command is printed on separate lines and follows the same sequence in which the commands were specified.
The semicolon metacharacter is useful when you need to run multiple commands sequentially on a single command line.

Asterisk *

The asterisk * metacharacter is used as a wildcard character to represent any sequence of characters, including none.

```
ls *.txt
```

In this example, *.txt is a wildcard pattern that matches any file in the current directory with a .txt extension. The ls command lists the names of all matching files.

Question mark `?`

The question mark `?` metacharacter is used as a wildcard character to represent any single character.

```
ls file?.txt
```

In this example, `file?.txt` is a wildcard pattern that matches any file in the current directory with a name starting with `file`, followed by any single character, and ending with the `.txt` extension.

Quoting

Quoting is a mechanism that allows you to remove the special meaning of characters, spaces, or other metacharacters in a command argument or shell script. You use quoting when you want the shell to interpret characters literally.

Symbol	Meaning
`\`	Escape metacharacter interpretation
`" "`	Interpret metacharacters within string
`' '`	Escape all metacharacters within string

Backslash `\`

The backslash character is used as an escape character. It instructs the shell to preserve the literal interpretation of special characters such as space, tab, and `$`. For example, if you have a file with spaces in its name, you can use backslashes followed by a space to handle those spaces literally:

```
touch file\ with\ space.txt
```

Double quotes `" "`

When a string is enclosed in double quotes, most characters are interpreted literally, but metacharacters are interpreted according to their special meaning. For example, you can access variable values using the dollar `$` character:

```
$ echo "Hello $USER"
Hello <username>
```

Single quotes `''`

When a string is enclosed in single quotes, all characters and metacharacters enclosed within the quotes are interpreted literally. Single quotes alter the above example to produce the following output:

```
$ echo 'Hello $USER'
Hello $USER
```

Notice that instead of printing the value of `$USER`, single quotes cause the terminal to print the string `"$USER"`.

Input/Output redirection

Symbol	Meaning
`>`	Redirect output to file, overwrite
`>>`	Redirect output to file, append
`2>`	Redirect standard error to file, overwrite
`2>>`	Redirect standard error to file, append
`<`	Redirect file contents to standard input

Input/output (IO) redirection is the process of directing the flow of data between a program and its input/output sources.
By default, a program reads input from *standard input*, the keyboard, and writes output to *standard output*, the terminal. However, using IO redirection, you can redirect a program's input or output to or from a file or another program.

Redirect output `>`

This symbol is used to redirect the standard output of a command to a specified file.

`ls > files.txt` will create a file called `files.txt` if it doesn't exist, and write the output of the `ls` command to it.
Warning: When the file already exists, the output overwrites all of the file's contents!

Redirect and append output `>>`

This notation is used to redirect and append the output of a command to the end of a file. For example,

`ls >> files.txt` *appends the output of the* `ls` *command to the end of file* `files.txt`*, and preserves any content that already existed in the file.*

Redirect standard output `2>`

This notation is used to redirect the standard error output of a command to a file. For example, if you run the ls command on a non-existing directory as follows,

`ls non-existent-directory 2> error.txt` *the shell will create a file called* `error.txt` *if it doesn't exist, and redirect the error output of the* `ls` *command to the file.*
Warning: When the file already exists, the error message overwrites all of the file's contents!

Append standard error `2>>`

This symbol redirects the standard error output of a command and appends the error message to the end of a file without overwriting its contents.

`ls non-existent-directory 2>> error.txt` *will append the error output of the* `ls` *command to the end of the* `error.txt` *file.*

Redirect input `<`

This symbol is used to redirect the standard input of a command from a file or another command. For example,

`sort < data.txt` *will* `sort` *the contents of the* `data.txt` *file.*

Command Substitution

Command substitution allows you to run command and use its output as a component of another command's argument. Command substitution is denoted by enclosing a command in either backticks (`**command**`) or using the `$()` syntax. When the encapsulate command is executed, its output is substituted in place, and it can be used as an argument within another command. This is particularly useful for automating tasks that require the use of a command's output as input for another command.
For example, you could store the path to your current directory in a variable by applying command substitution on the `pwd` command, then move to another directory, and finally

return to your original directory by invoking the `cd` command on the variable you stored, as follows:

```
$ here=$(pwd)
$ cd path_to_some_other_directory
$ cd $here
```

Command Line Arguments

Command line arguments are additional inputs that can be passed to a program when the program is run from a command line interface. These arguments are specified after the name of the program, and they can be used to modify the behavior of the program, provide input data, or provide output locations. Command line arguments are used to pass arguments to a shell script.
For example, the following command provides two arguments, arg1, and arg2, that can be accessed from within your Bash script:

```
$ ./MyBashScript.sh arg1 arg2
```

Summary

In this reading, you learned that:

- Metacharacters such as `#`, `;`, `*`, and `?` are characters that the shell interprets with special meanings
- Quoting allows you to ensure any special characters, spaces, or other metacharacters are interpreted literally by the shell
- Input/output redirection redirects a program's input or output to/from a file
- Command substitution allows you to use the output of a command as an argument for another command
- Command line arguments can be used to pass information to a shell script

Reading: Introduction to Advanced Bash Scripting

Conditionals

Conditionals, or `if` statements, are a way of telling a script to do something only under a specific condition.
Bash script conditionals use the following `if` - `then` - `else` syntax:

```
if [ condition ]
then
    statement_block_1
else
    statement_block_2
fi
```

If the `condition` is `true`, then Bash executes the statements in `statement_block_1` before exiting the conditional block of code. After exiting, it will continue to run any commands after the closing `fi`.

Alternatively, if the `condition` is `false`, Bash instead runs the statements in `statement_block_2` under the `else` line, then exits the conditional block and continues to run commands after the closing `fi`.

Tips:

- *You must always put spaces around your condition within the square brackets `[ ]`.*
- *Every `if` condition block must be paired with a `fi` to tell Bash where the condition block ends.*
- *The `else` block is optional but recommended. If the condition evaluates to `false` without an `else` block, then nothing happens within the `if` condition block. Consider options such as echoing a comment in `statement_block_2` to indicate that the condition was evaluated as `false`.*

In the following example, the condition is checking whether the number of command-line arguments read by some Bash script, `$#`, is equal to 2.

```
if [[ $# == 2 ]]
then
  echo "number of arguments is equal to 2"
else
  echo "number of arguments is not equal to 2"
fi
```

Notice the use of the double square brackets, which is the syntax required for making integer comparisons in the condition `[[ $# == 2 ]]`.

You can also make string comparisons. For example, assume you have a variable called `string_var` that has the value `"Yes"` assigned to it. Then the following statement evaluates to `true`:

```
1. `[ $string_var == "True" ]`
```

Notice you only need single square brackets when making string comparisons.

You can also include multiple conditions to be satified by using the "and" operator `&&` or the "or" operator `||`. For example:

```
1. if [ condition1 ] && [ condition2 ]
2. then
3.     echo "conditions 1 and 2 are both true"
4. else
5.     echo "one or both conditions are false"
6. fi
```

```
1. if [ condition1 ] || [ condition2 ]
2. then
3.     echo "conditions 1 or 2 are true"
4. else
5.     echo "both conditions are false"
6. fi
```

Logical operators

The following logical operators can be used to compare integers within a condition in an `if` condition block.
`==`: is equal to
If a variable `a` has a value of 2, the following condition evaluates to `true`; otherwise it evalutes to `false`.

```
1. $a == 2
```

`!=`: is not equal to
If a variable `a` has a value different from 2, the following statement evaluates to `true`. If its value is 2, then it evalutes to `false`.

```
a != 2
```

***Tip**: The `!` logical negation operator changes `true` to `false` and `false` to `true`.*

`<=`: is less than or equal to

If a variable `a` has a value of 2, then the following statement evaluates to `true`:

```
a <= 3
```

and the following statement evalutates to `false`:

```
a <= 1
```

Alternatively, you can use the equivalent notation `-le` in place of `<=`:

```
a=1
b=2
if [ $a -le $b ]
then
   echo "a is less than or equal to b"
else
   echo "a is not less than or equal to b"
fi
```

We've only provided a small sampling of logical operators here. You can explore resources such as the Advanced Bash-Scripting Guide to find out more.

Arithmetic calculations

You can perform integer addition, subtraction, multiplication, and division using the notation `$(())`.

For example, the following two sets of commands both display the result of adding 3 and 2.

```
echo $((3+2))
```

or

```
a=3
b=2
c=$(($a+$b))
echo $c
```

Bash natively handles integer arithmetic but does not handle floating-point arithmetic. As a result, it will always truncate the decimal portion of a calculation result.

For example:

```
1. echo $((3/2))
```

prints the truncated integer result, `1`, not the floating-point number, `1.5`.
The following table summarizes the basic arithmetic operators:

Symbol	Operation
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division

*Table: **Arithmetic operators***

Arrays

The **array** is a Bash built-in data structure. An array is a space-delimited list contained in parentheses.ʊTo create an array, declare its name and contents:

```
1. my_array=(1 2 "three" "four" 5)
```

This statement creates and populates the array `my_array` with the items in the parentheses: `1`, `2`, `"three"`, `"four"`, and `5`.
You can also create an empty array by using:

```
1. declare -a empty_array
```

If you want to add items to your array after creating it, you can add to your array by appending one element at a time:

```
1. my_array+=("six")
2. my_array+=(7)
```

This adds elements `"six"` and `7` to the array `my_array`.
By using indexing, you can access individual or multiple elements of an array:

```
# print the first item of the array:
echo ${my_array[0]}

# print the third item of the array:
echo ${my_array[2]}

# print all array elements:
echo ${my_array[@]}
```

Tip: *Note that array indexing starts from 0, not from 1.*

`for` loops

You can use a construct called a `for` loop along with indexing to iterate over all elements of an array.
For example, the following `for` loops will continue to run over and over again until every element is printed:

```
for item in ${my_array[@]}; do
  echo $item
done
```

or

```
for i in ${!my_array[@]}; do
  echo ${my_array[$i]}
done
```

The `for` loop requires a `; do` component in order to cycle through the loop. Additionally, you need to terminate the `for` loop block with a `done` statement.
Another way to implement a `for` loop when you know how many iterations you want is as follows. For example, the following code prints the number 0 through 6.

```
N=6
for (( i=0; i<=$N; i++ )) ; do
  echo $i
done
```

You can use `for` loops to accomplish all sorts of things. For example, you could count the number of items in an array or sum up its elements, as the following Bash script does:

```
#!/usr/bin/env bash
# initialize array, count, and sum
my_array=(1 2 3)
count=0
sum=0
for i in ${!my_array[@]}; do
  # print the ith array element
  echo ${my_array[$i]}
  # increment the count by one
  count=$(($count+1))
  # add the current value of the array to the sum
  sum=$(($sum+${my_array[$i]}))
done
echo $count
echo $sum
```

Go ahead and try running this script, so you get a sense of how this loop works.

Summary

In this lab, you learned that:

- Conditional statements can be used to run commands based on whether a specified condition is `true`
- Logical operators do `true` / `false` comparisons
- Arithmetic operators perform basic arithmetic calculations
- You can create list-like arrays and access their individual elements
- `for` loops execute operations repeatedly, based on a looping index

4] Scheduling Jobs using Cron

Job scheduling: -

- **Schedule jobs to run automatically at certain times.**

  Load script at midnight every night

Backup script to run every Sunday at 2 AM

- Cron allows you to automate such tasks

What are cron, crond, and crontab?

- Cron is a service that runs jobs
- Crond interprets 'crontab files' and submits jobs to cron.
- A crontab is a table of jobs and schedule data
- Crontab command invokes text editor to edit a crontab file.

Scheduling cron jobs with crontab –

```
$ crontab -e    # opens editor
```

Job syntax: -

m h dom mon dow command

where →

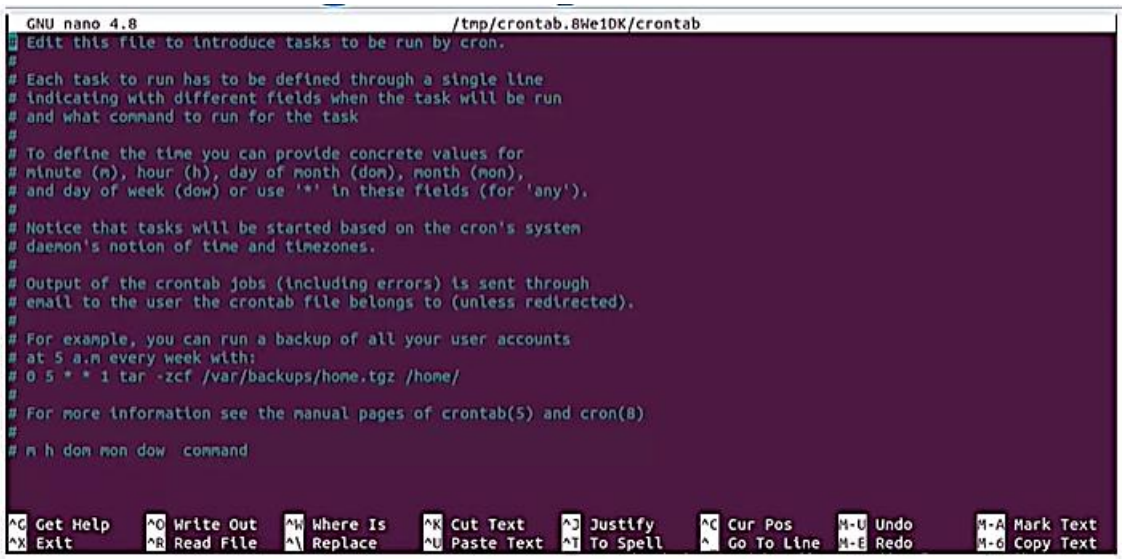
m – minute
h – hour
dom – day of month
mon – month
dow – day of week

Example job:

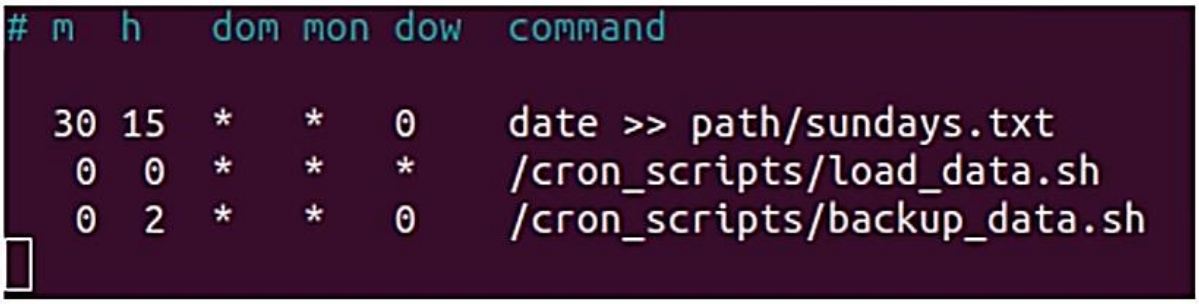
30 15 * * 0 date >> Sundays.txt

→ Append the current date to the file 'sundays.txt' at 15:30 every Sunday.

Closing the editor and saving the changes adds the job to the cron table.

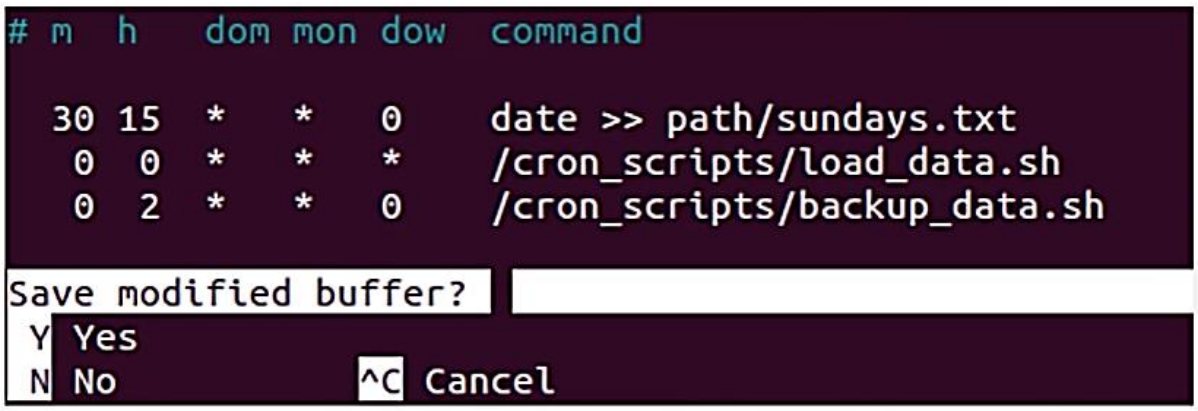


Entering jobs –



- The first entry instructs cron to append the current date to the file 'sundays.txt,' at 15:30 every Sunday.
- The next line specifies a "load data" shell script to run at midnight every day, while
- The last line results in cron running the "backup" data shell script to run at 2 AM on Sundays.

Exit text editor and save –



- To save the job, first type "control x" to exit the editor, and then enter "y" to save your changes.

Viewing and removing cron jobs –

```
jgrom@GROOT617:~$ crontab -l  | tail -6
#
# m  h   dom mon dow  command

  30 15  *   *   0    date >> path/sundays.txt
   0  0  *   *   *    /cron_scripts/load_data.sh
   0  2  *   *   0    /cron_scripts/backup_data.sh
jgrom@GROOT617:~$
```

```
$ crontab -e  # add/remove cron job with editor
```

- Running crontab with the "l" option returns a list of all cron jobs and their schedules.
- Used "tail" to avoid returning all of the comments from the crontab file.
- To remove a job, simply invoke the crontab editor, delete the corresponding line in the crontab file, and save the changes.

Summary: -

- Cron jobs can be scheduled to run periodically at selected times.
- 'm h dom mon dow command' is the cron job syntax.
- Cron jobs can be edited running 'crontab -e.'
- 'crontab -l' lists all cron jobs in the cron table.

Summary & Highlights: -

- A shell script is an executable text file that begins with a 'shebang' interpreter directive
- Shell scripts are used to run commands and programs and can interpret command line arguments
- Filters are shell commands, and the pipe operator allows you to chain filter commands
- Shell variables can be assigned values with '=' and listed using 'set'
- Environment variables are shell variables with extended scope; create with 'export,' list with 'env'
- Jobs can be scheduled to run periodically at selected times
- 'm h dom mon dow command' is the cron job syntax
- Command substitution is used to replace a command with its output
- The Bash shell-scripting feature 'concurrent mode' allows commands to run in parallel

Linux and Bash Command Cheat Sheet: The Basics

Getting information

Return your user name
whoami

Return your user and group id
id

Return operating system name, username, and other info
uname -a

Display reference manual for a command
man top

List available man pages, including a brief description for each command:
man -k

Get help on a command
curl --help

Return the current date and time
date

Navigating and working with directories

List files and directories by date, newest last
ls -lrt

Find files in directory tree with suffix 'sh'
find -name '*.sh'

Return present working directory
pwd

Make a new directory
mkdir new_folder

Change the current directory:
up one level
cd ../
home
cd ~ or cd
or some other path
cd another_directory

Remove directory, verbosely
rmdir temp_directory -v

Monitoring performance and status

List selection of or all running processes and their PIDs
ps
ps -e

Display resource usage
top

List mounted file systems and usage
df

Creating, copying, moving, and deleting files

Create an empty file or update existing file's timestamp

touch a_new_file.txt

Copy a file

cp file.txt new_path/new_name.txt

Change file name or path

mv this_file.txt that_path/that_file.txt

Remove a file verbosely

rm this_old_file.txt -v

Working with files permissions

Change/modify file permissions to 'execute' for all users
chmod +x my_script.sh

Change/modify file permissions to 'execute' only for you, the current user

chmod u+x my_file.txt

Remove 'read' permissions from group and other users

chmod go-r

Displaying/Printing file and string contents

Display file contents
cat my_shell_script.sh

Display file contents page-by-page
more ReadMe.txt

Display first N lines of file
head -10 data_table.csv

Display last N lines of file
tail -10 data_table.csv

Display string or variable value
echo "I am not a robot"
echo "I am $USERNAME"

Basic text wrangling

Sorting lines and dropping duplicates

Sort and display lines of file alphanumerically
sort text_file.txt
In reverse order
sort -r text_file.txt
Drop consecutive duplicated lines and display result

uniq list_with_duplicated_lines.txt

Displaying basic stats

Display the count of lines, words, or characters in a file
Lines:
wc -l table_of_data.csv
Words:
wc -w my_essay.txt
Characters:
wc -m some_document.txt

Extracting lines of text containing a pattern

Some frequently used options for grep:

Option	Description
-n	Print line numbers along with matching lines
-c	Get the count of matching lines
-i	Ignore the case of the text while matching
-v	Print all lines which do not contain the pattern
-w	Match only if the pattern matches whole words

Return lines matching a pattern from files matching a filename pattern - case insensitive and whole words only
Extract lines containing the word "hello", case insensitive and whole words only
grep -iw hello *.txt

Return file names with lines matching the pattern 'hello' from files matching a filename pattern
Extract lines containing the pattern "hello" from all files in the current directory ending in .txt
grep -l hello *.txt

Merge two or more files line-by-line, aligned as columns

Use paste to align file contents into a Tab-delimited table, one row for each customer

paste first_name.txt last_name.text phone_number.txt

Use a comma as a delimiter instead of the default Tab delimiter

paste -d "," first_name.txt last_name.text phone_number.txt

Use the cut command to extract a column from a table-like file

Extract first names, line-by-line

```
cut -d "," -f 1 names.csv
```

Extract the second to fifth characters (bytes) from each line of a file

```
cut -b 2-5 my_text_file.txt
```

Extract the characters (bytes) from each line of a file, starting from the 10th byte to the end of the line

```
cut -b 10- my_text_file.txt
```

Compression and archiving

Archive a set of files

```
tar -cvf my_archive.tar.gz file1 file2 file3
```

Compress a set of files

```
zip my_zipped_files.zip file1 file2
zip my_zipped_folders.zip directory1 directory2
```

Extract files from a compressed zip archive

```
unzip my_zipped_file.zip
unzip my_zipped_file.zip -d extract_to_this_direcory
```

Performing/ Working with network operations

Print hostname

```
hostname
```

Send packets to URL and print response

```
ping www.google.com
```

Display or configure system network interfaces

```
ifconfig
ip
```

Display contents of file at a URL

```
curl <url>
```

Download file from a URL

```
wget <url>
```

Bash shebang

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

Get the path to a command

```
which bash
```

Pipes, filters, and chaining

```
# Chain filter commands using the pipe operator
ls | sort -r

# Pipe the output of manual page for ls to head to display the first 20 lines
man ls | head -20

# Use a pipeline to extract a column of names from a csv and drop duplicate names

cut -d "," -f1 names.csv | sort | uniq
```

Working with Shell and Environment Variables

```
# List all shell variables
set

# Define a shell variable called my_planet and assign value Earth to it
my_planet=Earth

# Display shell variable
echo $my_planet

# Reading user input into a shell variable at the command line

read first_name
```

Tip: *Whatever text string you enter after running this command gets stored as the value of the variable* `first_name`

```
# List all environment variables
env

# Environment vars: define/extend variable scope to child processes
export my_planet
export my_galaxy='Milky Way'
```

Metacharacters

Comments
The shell will not respond to this message

Command separator
echo 'here are some files and folders'; ls

File name expansion wildcard
ls *.json

Single character wildcard
ls file_2021-06-??.json

Quoting

Single quotes - interpret literally
echo 'My home directory can be accessed by entering: echo $HOME'

Double quotes - interpret literally, but evaluate metacharacters
echo "My home directory is $HOME"

Backslash - escape metacharacter interpretation
echo "This dollar sign should render: \$"

I/O Redirection

Redirect output to file and overwrite any existing content
echo 'Write this text to file x' > x

Append output to file
echo 'Add this line to file x' >> x

Redirect standard error to file
bad_command_1 2> error.log

Append standard error to file
bad_command_2 2>> error.log

# Redirect file contents to standard input
$ tr “[a-z]” “[A-Z]” < a_text_file.txt

The input redirection above is equivalent to
$cat a_text_file.txt | tr “[a-z]” “[A-Z]”

Command Substitution

# Capture output of a command and echo its value
THE_PRESENT=$(date)
echo "There is no time like $THE_PRESENT"

Capture output of a command and echo its value

```
echo "There is no time like $(date)"
```

Command line arguments

```
./My_Bash_Script.sh arg1 arg2 arg3
```

Batch vs. concurrent modes

Run commands sequentially
```
start=$(date); ./MyBigScript.sh ; end=$(date)
```

Run commands in parallel
```
./ETL_chunk_one_on_these_nodes.sh & ./ETL_chunk_two_on_those_nodes.sh
```

Scheduling jobs with Cron

Open crontab editor
```
crontab -e
```

Job scheduling syntax
```
m h dom mon dow command
```
minute, hour, day of month, month, day of week

Tip: *You can use the* `*` *wildcard to mean "any".*

Append the date/time to file every Sunday at 6:15 pm
```
15 18 * * 0 date >> sundays.txt
```

Run a shell script on the first minute of the first day of each month
```
1 0 1 * * ./My_Shell_Script.sh
```

Back up your home directory every Monday at 3 am
```
0 3 * * 1 tar -cvf my_backup_path\my_archive.tar.gz $HOME\
```

Deploy your cron job
Close the crontab editor and save the file

List all cron jobs
```
crontab -l
```

Week 4

Final Project

Module 1: - Final Project

Conditionals

Shell Script: Conditionals

This reading will get you sufficiently familiar with bash *conditionals* for the final project.

Conditionals are ways of telling a script to do something *under specific condition(s)*.

In this reading, you will learn about shell script conditionals using if else.

If

Syntax:

```
if [ condition ]
then
   statement
fi
```

You must always put spaces around your conditions in the [ ].

Every if condition block must be paired with a fi.

Example

```
$ cat if_example.sh
a=1
b=2
if [ $a -lt $b ]
then
   echo "a is less than b"
fi
```

```
$ sh if_example.sh  # sh tells the terminal to run the script if_example.sh using the default shell
a is less than b
```

If-Else

Syntax:

```
if [ condition ]
then
   statement_1
else
   statement_2
```

```
fi
```
You don't use then for else cases.

Example

```
$ cat if_else_example.sh
a=3
b=2
if [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "a is greater than or equal to b"
fi

$ sh if_else_example.sh
a is greater than or equal to b
```

Elif

The statement elif means "else if":

Syntax:

```
if [ condition_1 ]
then
   statement_1
elif [ condition_2 ]
then
   statement_2
fi
```

Example

```
$ cat elif_example.sh
a=2
b=2
if [ $a -lt $b ]
then
   echo "a is less than b"
elif [ $a == $b ]
then
   echo "a is equal to b"
else # Here a is not <= b, so a > b
   echo "a is greater than b"
fi

$ sh elif_example.sh
a is equal to b
```

Nested Ifs

As in other prgramming languages, it's also possible to nest if-statements.

Syntax:

```
if [ condition_1 ]
then
   statement_1
elif [ condition_2 ]
   statement_2
   if [ condition_2.1 ]
   then
      statement_2.1
   fi
else
   statement_3
fi
```

Example

```
$ cat nested_ifs_example.sh
a=3
b=3
c=3
if [ $a == $b ]
then
   if [ $a == $c ]
   then
      if [ $b == $c ]
      then
         echo "a, b, and c are equal"
      fi
   fi
else
   echo "the three variables are not equal"
fi

$ sh nested_ifs_example.sh
a, b, and c are equal
```

Alternatively, this example could have been simplified to a single if-statement:

```
a=3
b=3
c=3
if [ $a == $b ] && [ $a == $c ] && [ $b == $c ]
then
   echo "a, b, and c are equal"
else
   echo "the three variables are not equal"
```

```
fi
```
&& means "and"

Bonus: "test"

Sometimes, instead of using brackets around conditions, you'll see the test command in use:

Example

```
$ cat test_example.sh
a=1
b=2
if test $a -lt $b
then
   echo "a is less than b"
fi

$ sh test_example.sh
a is less than b
```

test and [ ] are the same command. We encourage using [ ] instead as it's more readable.