Course Overview

Course Objective: Develop your Linux Command Line & Shell Scripting skills and advance your career

Basic Admin

- **Installing Linux on a Virtual Machine**
- **Setting up Accounts**
- **Setting up Permissions**
- **Unix Commands - A Comprehensive Primer**

Basics of Shell Scripting

- **Overview**
- **Editors - Vim and Notepad++**
- **Shell Expansion**
- **Quotes**
- **Braces, Brackets and Parentheses**
- **A starter Shell script**
- **Input and Output Processing**

Course Overview

Core Processing

- Iterations
  - For loops
  - While loops
  - Until loops

- IF statements
- Case statements
- Regular Expressions
- Arrays
- Dates
- Functions

Course Overview

File Processing

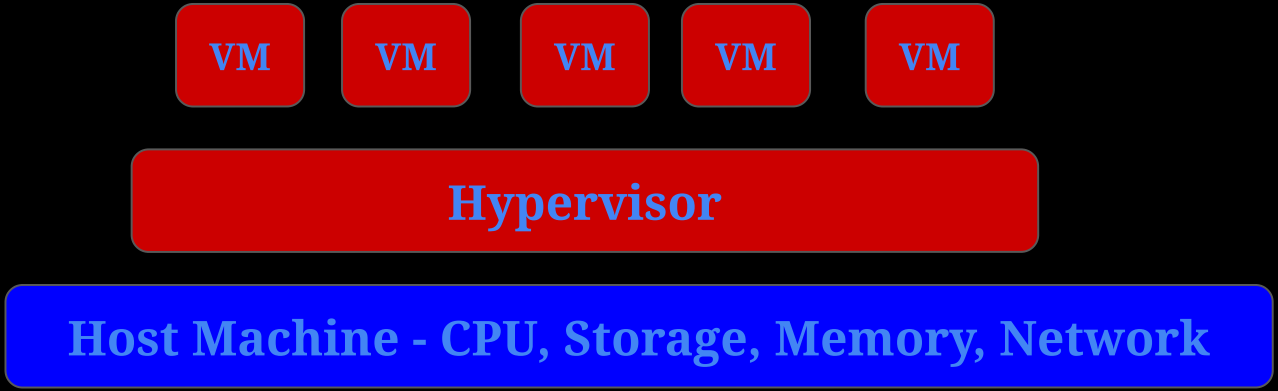
- **Cut**
- **Sort**
- **Uniq**
- **Awk**
- **Grep**
- **Sed**

Cron - Job Scheduling

Projects

- **Document Merge**
- **Listing Users and Groups**
- **Cron - Scheduling a backup**
- **Aggregating data in CSV files**
- **Pivoting input data**
- **Formatting output data**
- **An Enterprise-level shell script putting it all together**

Virtual Environment



Virtual Machine - A computing resource that uses software instead of a physical computer to run programs and deploy apps

Each VM in encapsulated from other VMs

Hypervisor - It is a program used to manage and run multiple VMs on a computer

Linux in a Virtual Environment

We install Linux in a Virtual Environment mainly for the following reasons:

- **Provides an Isolated Environment - for eg. if you install Linux Mint (a Linux distribution) in a Virtual Environment such as Oracle VirtualBox, on a Windows system, it minimizes setup conflicts. In case something messes up with your Linux install it will not affect your Windows system. You can just delete your Virtual Machine and start all over again.**

- **Easier to clone a Virtual Machine in Virtual Box rather than re-create multiple instances of Linux distros.**

- **Provides an easy setup mechanism for students wanting to set up a bare bone system on which they could practice Unix Admin & Shell Scripting without affecting other systems.**

Linux in a Virtual Environment ...

What we need to do to set up Linux in a Virtual Environment:

1. Install Oracle VirtualBox - https://www.virtualbox.org/wiki/Downloads

   It provides a shell with a UI for managing your different Virtual Machines. So you could have multiple VMs in VirtualBox for different Linux distros - one for Centos, one for Linux Mint etc. Each one of them can be managed independently.

2. Download Linux Mint - https://www.linuxmint.com/download.php

   This is one of the more popular Linux distros.

3. Install Linux Mint on VirtualBox

4. Install any utility software. In our case, we are going to be installing Notepad++ so you have an alternative to using the command-key based Vi editor.

Account Management

- **Overview of Topics covered:**

    Types of Users - Root (Sudo), Admin and Regular Users

        What is a Root User? What are the privileges?
        How do you log in as Root?
        Why should you not log in as Root?
            What is Sudo? Why use Sudo instead of Root?
        How do you lock/unlock Root access?

        What is an Admin user in Linux Mint? What is the benefit of this role?

        What is a Regular User? What are the privileges of a Regular User?

        How do you give a Regular User Sudo privileges?

Account Management

- **Overview of Topics covered (contd.):**

  User Management

  - **Adding Users**
  - **Adding Groups**
  - **Where can you see the stored User and Group information?**

  - **Adding Users to Groups**
  - **Adding Users to Sudo**

  - **Deleting Users from Groups**
  - **Deleting Groups**
  - **Deleting Users**

Account Management

- What is a Root User? What are the privileges?

  Default user with full privileges
  Can add/modify/delete any file
    Can run any executable
    Can add/delete Accounts and modify Permissions
    Is defined by a user id of 0

- How do you log in as Root?

    The administrator assigns a password to Root
    Use 'su' or 'su -'
  The prompt will show a pound sign instead of a dollar to distinguish root
    The user id for Root should be 0

Account Management

- **Why should you not log in as Root?**

  Running commands as Root could potentially cause irreversible damage to the system:

  You could accidentally delete or modify system files
  A malicious program could log in as Root and cause severe damage
  Anyone could log in as Root once they know the password

- **What is Sudo?**

  Sudo is short form for SuperUser Do. It gives you the ability to act as Root for 'One' command at a time.

Account Management

- **Why use Sudo instead of Root?**

  Since you have to run Sudo for each command you are constantly reminded that you are running commands as root. It is very unlikely that you will accidentally run commands causing damage after repeated reminders. Whereas you could be logged in as Root and forget about it. Inadvertently you could run many commands in sequence that could cause system wide damage and not realize it. You would need a system reinstall in that case.

  When you log in as Root the user has to log in with the Root user id. You cannot track which user ran a certain command. In case of Sudo the user has to log in with their own user id. This way we can track user activity.

Account Management

- How do you lock Root access?

  ```
  sudo passwd -l root
  ```

- How do you unlock Root access?

  ```
  sudo passwd root
  ```

- What is an Admin user in Linux Mint? What is the benefit of this role?

  By default the Admin is the user that installed the system. The Admin has sudo access by default and can assign the password for the Root user, making Root active.
  The benefit of the role is that by default you are starting off with sudo access and Root access is optional and can be kept permanently disabled. In some other versions of Linux, the Root user is the one that installed the system and is the default user. So you could be logged in as Root for a while, simply since that is your default role. That could potentially cause a lot of problems.

Account Management

- What is a Regular or Standard user? What are the privileges of a Regular user?

  The Regular or Standard user has the ability to perform routine tasks such as reading and writing files, and running executables. The scope of the Regular user is limited by its permissions.

- How do you give a Regular User Sudo privileges?

  Add the user to the Sudo group using either of the following commands:

  ```
  sudo adduser <username> sudo
  sudo usermod -a -G sudo <user>
  ```

- Adding Users

  ```
  sudo adduser test_user
  ```

- Adding Groups

  ```
  sudo addgroup test_main_group
      sudo addgroup test_secondary_group
  ```

Account Management

- Where can you see the stored User and Group information?

  User information can be viewed in the /etc/passwd file
  Group information can be viewed in the /etc/group file

- What are the contents of the /etc/passwd file?

  1) Username
  2) Password: 'x' means that the password is encrypted password and is stored in /etc/shadow file
  3) User Id (UID): UID 0 (zero) is root. UIDs 1-999 are reserved for system accounts etc. Regular ids start at 1000
  4) Group Id (GID): The primary group Id. This is stored in the /etc/group file
  5) User Info: Information about the user - full name etc
  6) Home directory: The login directory of the user. If this field is blank, the login directory for the user is /
  7) Command/shell: The shell path or command

Account Management

- Adding Users to Groups
  Add a user to the main group:
  ```
  sudo adduser test_user test_main_group
  ```
  Add a user to the main group and make it the primary group for the user:
  ```
  sudo usermod -g test_main_group test_user
  ```
  Add a user to the secondary group
  ```
  sudo usermod -aG test_secondary_group test_user
  ```

- Adding User to Sudo group
  ```
  sudo adduser <username> sudo
  ```
  or
  ```
  sudo usermod -a -G sudo <user>
  ```

- List users for a group
  ```
  groups test_user
  ```

Account Management

- Deleting Users from Groups

```
sudo deluser test_user test_secondary_group
```

- Deleting Groups

```
sudo delgroup test_secondary_group
```

- Deleting Users

```
sudo  deluser -r test_user
```

Permissions

- Overview of Permissions on Files/Folders:

    - Adding/Changing/Removing Permissions

    - Using chmod rwx
        - Using chmod with numerical values

    - Change Ownership - chown
    - Change Group - chgrp

    - Miscellaneous

    - Copying Permissions
        - Applying Permissions recursively

Permissions

- **Overview of Permissions (contd.):**
    - **Permissions on:**
        - **User, Group & Other**
        - **Directories, Files & Links**
    - **Directory vs File Permissions**

Permissions

- Adding/Changing/Removing Permissions

The permissions on a file / directory are specified as:

<d/-/l> rwx (User) rwx (Group) rwx (Other)
d = Directory
- = Filename
l = Link

r - Read permissions
w - Write permissions
x - Execute permissions

User - The owner of the file/directory
Group - All users in the Group have access to the file/directory as defined by the
permissions

Permissions

- Adding/Changing/Removing Permissions (contd.)

In case there are no permissions a (-) replaces r (read), w (write) or x (execute)

The sample permissions for a directory with full permissions for User, Group and Other :

drwxrwxrwx

The sample permissions for a file with full permissions for User, Group and Other :

-rwxrwxrwx

A directory with Permissions such as User (Read, Write, Execute), Group (Read, Write)
and Other (No permissions) would be defined as:

Permissions

- **Adding/Changing/Removing Permissions (contd.)**

Use the chmod command to add read <r>, write <w>, execute <x> permissions on a directory / file as below:

```
chmod <u/g/o> + rwx <directory / filename>
…where u - user, g - group, o - other
```

Consider a file that has zero permissions for User, Group and Other:

```
- --- --- ---
```

To set the permissions as User (Read, Write), Group (Read, Write), Other (Read) run the command:

Permissions

- **Adding/Changing/Removing Permissions (contd.)**

To remove permissions use the - sign instead of +. So to remove write permissions on the User and Group for the above scenario run:

```
chmod u-w,g-w <filename>
```

To change all (User, Group, Other) permissions use 'a' instead of 'u','g' and 'o':

```
chmod a+rwx <filename>
```

Permissions

- **Numerical Permissions**

  The numerical permissions are set as below:

  R ($4 = 2^2$)
  W ($2 = 2^1$)
  X ($1 = 2^0$)

  The total permissions are got by adding R+W+X for user, group and other

  To get r-x rwx -- x you would use 5(r-x) 7(rwx) 1(--x)

  Instead of chmod u+rx,g+rwx,o+x you could use chmod 571

Permissions

- **Change Ownership - chown**

  The Ownership of a file/directory can be changed using chown:
  ```
  sudo chown <user> <file/directory>
  ```
  Use the -v option for verbose output:
  ```
  sudo chown -v <user> <file/directory>
  ```

- **Change Group Ownership - chgrp**

  The Group Ownership of a file/directory can be changed using chown:
  ```
  sudo chgrp <user> <file/directory>
  ```
  Use the -v option for verbose output:
  ```
  sudo chgrp -v <user> <file/directory>
  ```

Permissions

- Copying Permissions

  Permissions can be copied between files and directories

  ```
  sudo <chmod/chown/chgrp> --reference=file1 file2
  ```

- Changing Permissions Recursively

  Permissions can be changed recursively:

  ```
  sudo <chmod/chown/chgrp> -R <user> <file/directory>
  ```

Permissions

- Permissions on:

    Users, Groups & Other

        User permissions apply only to the owner of the file/directory
        Group permissions apply to all users within the group of the file/directory
        Other permissions apply to users that do not fall in either the User or Group category

    Directories

        read - Can list the contents of the directory
        write - Can create files/directories in the directory
        execute - Can switch to the directory

Permissions

- **Permissions on:**

  Files

  read - Can read the file
  write - Can write to the file
  execute - Can run the executable

  Links

  The Link displays full permissions but the actual permissions are determined by the file/directory it is pointing to

Permissions

- **Directory vs File Permissions**

  File permissions are more restrictive/permissive than Directory permissions and override the Directory permissions except in certain cases such as a Delete. For the case of Delete, the permissions of the directory take precedence

Unix Commands

- **Topics Covered:**
    - **Getting information on commands**
    - **Types of Unix commands:**
        - **Internal Shell commands (built-in)**
    - **External Shell commands**
    - **Alias commands**
    - **Function commands**
    - **Keyword commands**

Unix Commands

- **Topics Covered (contd.):**

  Categories:

  System Commands
  File and Directory Commands
  User Info Commands
  Networking Commands
  SSH and File Transfer Commands
  Disk Commands
  Security Commands
  Archiving Commands
  Search Commands
  Installation Commands

  ***** User Management Commands**

Unix Commands

- Getting information on commands

    Listing all commands

        compgen -c (lists all the commands you could run)
        compgen -a (lists all the aliases you could run)
        compgen -b (lists all the built-ins you could run)
        compgen -k (lists all the keywords you could run)
        compgen -A function (lists all the functions you could run)

    Getting help on commands

        <command> --help/h
        man
        tldr
        bropages

Unix Commands

- Types of Unix commands
    - Find the Type of command
        - type -t
        - type -a
    - Internal Shell commands (built-in)
    - External Shell commands (file)
    - Alias commands
    - Function commands
    - Keyword commands

Unix Commands

- **Internal Shell commands (built-in)**

  List all built-in commands using:

  ```
  help -d
  compgen -b
  ```

  Some common shell commands:

  cd - Change directory
  pwd - Display current path
  logout - Log out of the current session

  Some common shell commands are:

  ```
  syedaf@MintVM:~/test_scripts$ type -a cd
  cd is a shell builtin
  ```

Unix Commands

- **External Shell commands (file)**

These are commands which have their own file or binary and are usually located in /usr/bin or /bin

Some examples of External Shell commands are ‘cat’, ‘man’, ‘awk’

```
syedaf@MintVM:~/test_scripts$ type -t cat
file

syedaf@MintVM:~/test_scripts$ type -a cat
cat is /usr/bin/cat
cat is /bin/cat
```

Unix Commands

- **Alias commands**

These are shortcut commands which are used to provide a convenient way to run longer commands without having to type in the full list of options each time

For example, the command ‘ls -latr’ displays all the files in the directory in the long format sorted by time in the reverse order

We can create an alias for it (lf) as below:

```
syedaf@MintVM:~/test_scripts$ alias lf='ls -latr'
syedaf@MintVM:~/test_scripts$ lf
total 8
-rw-r--r--  1 syedaf test_main_group          0 Aug  6 17:05 test.sh
drwxrwxr-x  2 syedaf syedaf       4096 Aug  6 17:05 .
drwxr-x---  10 syedaf test_main_group 4096 Aug  7 09:45
```

Unix Commands

- Keyword commands

  Keyword commands are commands reserved for shell programming, such as 'if', 'then', 'else', 'for', 'while' etc

```
syedaf@Mint-VM:~$ type if then else for while
if is a shell keyword
then is a shell keyword
                    else is a shell keyword
for is a shell keyword
while is a shell keyword
```

Unix Commands

- **Function commands**

Bash functions allow grouping of a set of commands for execution with a single name. They differ from aliases in that they allow for control flow of execution.

A simple function for listing files with a certain extension could be defined as follows:

```
batchexec ()
{
        find . -type f -iname '*.'${1}''
}
```

It can be executed as below and will list shell scripts in the directory with a sh extension:

```
syedaf@MintVM:~/test_scripts$ batchexec sh
./test.sh
```

Unix Commands

- **Command Categories**

  System commands

hostname	**-- Name of the host**
whoami	**-- Your current user name**
who	**-- Who is logged into the system**
date	**-- The current date and time**
free -h	**-- Display free and used system memory**
top	**-- Display real-time info about running processes**
ps -ef	**-- Display all current running processes in the system**
kill pid	**-- Kill a certain process**
killall <p>	**-- Kill all processes named <p>**
ip a	**-- Display all network interfaces and ip addresses**
ping <host>	**-- Test connection to host**
head/tail -n /var/log/syslog	**-- Display first or last 'n' lines of the system log**

Unix Commands

- Command Categories (contd.) - File and Directory commands

ls -al	-- List all the files in a directory (long format)
rm -f <file>	-- Remove a file without confirmation
touch <file>	-- Create an empty file
cat <file>	-- Read the contents of the file
head -n <file>	-- Display the top <n> lines of the file
tail -n <file>	-- Display the last <n> lines of the file
tail -f <file>	-- Scroll and display the last 10 lines of the file
less <file>	-- Scroll through contents of file
mkdir <dir>	-- Make a directory
rm -rf <dir>	-- Delete the directory and it's contents
cd	-- Change to directory
pwd	-- Display present working directory
ln -s	-- ln -s <filepath> linkname
cp <file1> <file2>	-- Copy file1 to file
mv <file1> <file2>	-- Move file1 to file2 (directory)

Unix Commands

- **Command Categories (contd.) -**

  User Info commands

  id <user> -- Displays the User Id and Group Ids for the user
  groups <user> -- Displays the Groups the User belongs to
  finger <user> -- Displays detailed information about the user
  getent <db> <user> -- Retrieve user information from the system file - passwd, group etc
  lslogins -- Displays login information about users in the system
  users -- Displays users currently logged in to the system
  last -a Displays last logged in users on the

Unix Commands

- **Command Categories (contd.) -**

  Networking commands

ip a	**-- Display all the IP addresses**
ifconfig	**-- Display all the IP addresses**
netstat	**-- Display network related information**
ping	**-- Test if a device can be reached over the network**

Unix Commands

- Command Categories (contd.) -

SSH and File Transfer commands

ssh (Secure Shell) -- Connect to host
scp (Secure Copy Protocol) -- Secure copy file to remote machine. Linear copy.
Rsync (Remote Sync) -- Sync files between the local and remote systems. Copy
deltas with the ability to resume interrupted transfers.
ftp / sftp (Secure File Transfer Protocol) -- Connect to remote hostname. Allows for
remote directory file listings and file removal.

get/mget	-- Get file(s) from host
put/mput	-- Put file(s) on host

Note: The Scp and Rsync commands are similar but Rsync provides many more

Unix Commands

- **Command Categories (contd.) -**

  Disk commands

fdisk	**-- Add, delete or make changes to partitions**
cfdisk	**-- Similar to fdisk but with a GUI**
df -h	**-- Report disk usage on mounted filesystems in a human-readable format**
du -ah	**-- Report total disk usage for all files and directories in a human-readable format**
pydf	**-- Displays mount points and their disk usage**
lsblk	**-- Displays disks and their partitions in a graphical manner**
hwinfo	**-- General purpose hardware information**

Unix Commands

- **Command Categories (contd.) -**

  Security commands

  who -- Display information on who is logged in to the system
  w -- Displays more detailed information on logged in users
  last -- Display recent logins
  ufw -- Uncomplicated Firewall. Front-end for configuration of a firewall
  ip -- Display information on the network interfaces
  kill -- Kill a process
  passwd -- Change passwords
  pwck -- Checks for required files and directories in /etc/passwd /etc/shadow files

Unix Commands

- Command Categories (contd.) -

  Archiving commands

  tar -- Standard archiving utility
  tar cvf (compress) tar xvf (extract)

  ar -- Maintains groups of files in a single file. Replaced with tar

  gzip -- File compression utility for compressing single files
  gzip <filename> (compress) gunzip <filename> (extract)

  bzip2 -- Provides higher compression than gzip but at a slower rate
  bzip2 -z <filename> (compress) bzip2 -d <filename>

Unix Commands

- **Command Categories (contd.) -**

  Search commands

  find -- Finds a file in a given directory and sub-directories
  locate -- Locates a file in the database
  which -- Determines which version of a program or command will run and displays the path
  whereis -- Locates the binary, source, and manual page files for a command
  whatis -- Provides summary descriptions from the man page
  apropos -- Similar to whatis but finds all matches containing the term
  grep -- Search for a pattern in file

  Installation commands
  apt – Advanced Packaging Tool. Linux Mint and Debian systems

Vim Editor

- Vi is the command-based screen editor of choice for UNIX systems

- Vim is a superset of Vi and has enhanced functionality:

    - Multiple undo and redo
    - Windows splitting and tabs
    - Code highlighting
    - Macros

- Vi has 2 modes: Command mode and Insert mode
    - Command mode is used for copying, pasting, deleting and navigating text and is the default mode on entering the editor
    - Insert mode is for entering text and is entered by pressing the ‘i’ key
    - Revert back to Command mode from Insert mode by pressing the Esc key

Vim Editor

- Type vim <filename> to enter edit mode

- q - Quit (will prompt in case changes have been made)
- q! - Quit without being prompted for saving in case of changes

- x - Save and quit (same timestamp)
- wq - Save and quit
- wq! - Save and quit . Force write in case of read-only files.

Vim Editor ...

Writing out to a file

- **Difference between :w and :w!**
- **:w prompts when overwriting a read-only file**
- **:w! overwrites a file without prompting**

Vim Editor ...

- Moving Around
    - Use the Arrow keys and PgUp/PgDn keys
    - Move down

        j - Move down 1 line

        <n>j - J lines to the bottom
    - Move to the bottom

        G

        :$

Vim Editor ...

- Moving Around
  - Move up

    k - Move up 1 line

    <n>k - k lines to the top
  - Move to the top

    gg

    :1

Vim Editor ...

- Moving Around
    - Move right

        l - Move right 1 character

        <n>l - l characters to the right

    - Move left

        h - Move left 1 character

        <n>h - h characters to the left

Vim Editor ...

- Moving Around
  - ^ to move the cursor to the start of the current line
  - $ to move the cursor to the end of the current line
  - Page forward - Ctrl-F
  - Page backward - Ctrl-B

Vim Editor ...

- Inserting Text
    - a - Append text to the right of the cursor
    - A - Append text at the end of the line
    - i - Insert text at the left of the cursor
    - I - Insert text at the beginning of the line
    - o - Open a line below the cursor
    - O - Open a line above the cursor

Vim Editor ...

- Changing Text
    - cw - Change the word to the right of the cursor
    - cc - Replace a line

- Deleting Text
    - x - Deletes a character the cursor is on
    - dw - Position the cursor at the beginning of the word and delete the word
    - dd - Deletes the line
    - <n>dd - Deletes n lines

Vim Editor ...

- Copy and Paste
  - yy - Copy/Yank the line
  - p - Paste the line below the cursor
  - P - Paste the line above the cursor
  - <n>yy - Yanks <n> lines into the buffer
  - :1,$y - Yanks all lines into the buffer

The above Paste commands work with the Delete command ‘dd’ too which puts the deleted lines in a buffer.

Shell Expansion

- Expansion - The shell processes special characters contained in a command before it processes the entire command as a whole

- The different types of expansion are:

    - Brace expansion
    - Tilde expansion
    - Parameter expansion
    - Command substitution
    - Arithmetic expansion
    - Process substitution
    - Word splitting
    - Filename expansion
    - Quotes

Brace Expansion

- In brace expansion multiple text strings are contained within a pattern enclosed in braces
- The main benefit of this expansion is when creating directories. For example when creating a bunch of directories for a series of months for a year range, this saves us a lot of work since we could now do this instead:

```
        syedaf@MintVM:~/test_dir$ mkdir {2020..2022}-{01..03}

syedaf@MintVM:~/test_dir$ ls
2020-01  2020-02  2020-03  2021-01  2021-02  2021-03  2022-01  2022-02  2022-03
```

If you had a specific range of years (2020,2022) and months (01-04), you could do this:

```
syedaf@MintVM:~/test_dir$ mkdir {2020,2022}-{01,04}
syedaf@MintVM:~/test_dir$ ls
2020-01  2020-04  2022-01  2022-04
```

Tilde Expansion

- The tilde character (~) when used at the beginning of a word expands into the home directory of the named user. For example:

```
syedaf@MintVM:~/test_dir$ echo ~

/home/syedaf

syedaf@MintVM:~/test_dir$ echo ~/*D*

/home/syedaf/Desktop
/home/syedaf/Documents
/home/syedaf/Downloads
```

Parameter Expansion

- In Parameter expansion the value of a variable which is set is processed to give a new value. For example:

```
syedaf@MintVM:~/test_dir$ var="this is a test string"

syedaf@MintVM:~/test_dir$ echo $var
this is a test string

syedaf@MintVM:~/test_dir$ echo ${var:0:4}
this

syedaf@MintVM:~/test_dir$ echo ${#var}
21

syedaf@MintVM:~/test_dir$ echo "${var/this/that}"
that is a test string
```

Command Substitution

- In Command substitution the output of a command is used to replace the command. For example:

```
                    syedaf@MintVM:~/test_dir$ echo $(pwd)
              /home/syedaf/test_dir
      syedaf@MintVM:~/test_dir$ echo $(ls)
              2020-01 2020-04 2022-01 2022-04
```

  The output of a command can also be piped to another:

```
                    syedaf@MintVM:~/test_dir$ ls -l $(which mv)
      -rwxr-xr-x 1 root root 137744 Feb  7  2022 /usr/bin/mv
```

  A variable can be set to the value of a system command using command substitution:

```
              syedaf@MintVM:~/test_dir$ var=$(date)
      syedaf@MintVM:~/test_dir$ echo $var
              Fri Aug 12 11:26:29 AM EDT 2022
```

Arithmetic Expansion

- In Arithmetic expansion the arithmetic expression is evaluated and then substituted:

```
        syedaf@MintVM:~/test_dir$ y=$((2 + 2))
syedaf@MintVM:~/test_dir$ echo $y
        4
syedaf@MintVM:~/test_dir$ y=$((6 / 2))
syedaf@MintVM:~/test_dir$ echo $y
        3
```

A value of a variable can also be substituted as below:

```
syedaf@MintVM:~/test_dir$ y=1
syedaf@MintVM:~/test_dir$ y=$((y + 1))

syedaf@MintVM:~/test_dir$ echo $y
2
```

Process Substitution

- **Process substitution allows the input or output of a command to appear as file. This helps to minimize the number of operations when comparing files as shown below:**

  Comparing 2 files in reverse order:

```
syedaf@MintVM:~/test_dir$ diff <(cat test_file_1.txt) <(cat test_file_2.txt)
… <differences displayed>
```

  Reversing file 2 so they match in order:

```
syedaf@MintVM:~/test_dir$ diff <(cat test_file_1.txt) <(cat test_file_2.txt | sort test_file_2.txt)
syedaf@MintVM:~/test_dir$
```

  To perform the same operation would take multiple steps normally: 1.) Sort the output of file 2 and pipe it to a temp file. 2.) Compare this temp file with file 1 3.) Delete the temp file

Word Splitting

- When a string variable is expanded the default separator is either space, tab or newline. These are the boundaries used to separate words in the string which can now be listed as an array

  This is useful when parsing parameters passed in to a script:

```
                syedaf@MintVM:~/test_dir$ var="parm1 parm2 parm3"
syedaf@MintVM:~/test_dir$ for word in $var;do
        echo "Word: $word";
        done
```

  Output:

```
        Word: parm1
        Word: parm2
        Word: parm3
```

  The word splitting can be prevented by setting the default separator to null as: IFS=

Filename Expansion

- Wildcards are used to enable filename expansion. For example:
- * - Match a group of 0 or more characters

```
syedaf@MintVM:~/test_dir$ ls *.txt
0.txt  1.txt  2.txt  3.txt  4.txt  5.txt  6.txt  7.txt  8.txt  9.txt  test_file_1.txt  test_file_2.txt
```

- ? - Match exactly 1 character

```
syedaf@MintVM:~/test_dir$ ls 1.?xt
1.txt
```

- [..] - Matches any of the characters in range

```
syedaf@MintVM:~/test_dir$ ls [1-4].txt
1.txt  2.txt  3.txt  4.txt
```

Quotes

- Quotes are used to prevent unwanted expansions

```
syedaf@MintVM:~/test_dir$ echo $1000
000 … since $1 is considered a variable with an undefined value
```

To suppress all expansions use single quotes:

```
syedaf@MintVM:~/test_dir$ echo '$1000'
$1000

syedaf@MintVM:~/test_dir$ echo "$USER"
syedaf

syedaf@MintVM:~/test_dir$ echo '$USER'
$USER
```

Quotes...

- Double quotes are used to prevent unwanted expansions in the case of filenames or to preserve spaces in strings

```
    syedaf@MintVM:~/test_dir$ var="new file.txt"
syedaf@MintVM:~/test_dir$ cp 1.txt $var
    cp: target 'file.txt' is not a directory
syedaf@MintVM:~/test_dir$ cp 1.txt "$var"
syedaf@MintVM:~/test_dir$ ls
    0.txt   1.txt   2020-01   2020-04   2022-01   2022-04   2.txt   3.txt   4.txt   5.txt   6.txt   7.txt
8.txt
    9.txt  'new file.txt'   test_file_1.txt   test_file_2.txt

syedaf@MintVM:~/test_dir$ echo one                two
    one two
syedaf@MintVM:~/test_dir$ echo "one               two"
    one                two
```

Quotes...

- Putting any command between backquotes causes the command to be executed:

- echo `date` is equivalent to
- echo $(date)

```
syedaf@MintVM:~$ echo $(date)
        Mon Sep 26 04:27:49 PM EDT 2022

syedaf@MintVM:~$ echo `date`
        Mon Sep 26 04:27:58 PM EDT 2022
```

Basic Shell Script

- Shebang
- Variables
- Command-line parameters
- Piping commands
- Child scripts
- Variable scope
- Exit code
- Comments

Shebang

- The first line in the shell script is called the shebang. It could be either #!/bin/bash or #!/bin/sh. It specifies the shell that the interpreter is supposed to use to run the commands in the script. #!/bin/bash uses the Bash shell and #!/bin/sh is a symbolic link which could point either to Bash or some other shell such as Dash on recent versions of Ubuntu

Variables

- A variable is a storage for a piece of information. It represents either a string or a numeric value
- System variables such as the present working directory (PWD) should be in upper-case. User defined variables should preferably be kept in lower-case since variables are case-sensitive and this avoids overwriting system variables by accident
- Assignment takes place with a name
- The value is displayed by preceding the variable with the $ sign
- There should be no spacing between the = and the variable and the value.
    - The shell script will throw an error since it treats the variable as a command

Variables...

- Special script variables
  - $0 - The name of the Bash script
  - $1 - $9 - The first 9 arguments to the Bash script. (As mentioned above)
  - $# - How many arguments were passed to the Bash script
  - $@ - All the arguments supplied to the Bash script
  - $? - The exit status of the most recently run process
  - $$ - The process ID of the current script

Variables...

- Enclose the contents of the variable in single or double quotes to prevent word splitting
- The difference between double and single quotes is that double quotes expands the enclosed variable while single quotes translate the enclosed variable literally

```
	syedaf@MintVM:~/test_dir$ var1="Test1"
syedaf@MintVM:~/test_dir$ echo $var1
Test1
syedaf@MintVM:~/test_dir$ var2="var2 and $var1"
syedaf@MintVM:~/test_dir$ echo $var2
var2 and Test1
syedaf@MintVM:~/test_dir$ var2='var2 and $var1'
syedaf@MintVM:~/test_dir$ echo $var2
var2 and $var1
```

Tip: Test out the output with an echo statement on the command line

Variables...

When quoting variables which include wildcard expansion do not quote the variable since the expanded value will be based on the resultant string including the quotes

```
syedaf@MintVM:~/test_dir$ var1="*"
syedaf@MintVM:~/test_dir$ echo ${var1}
0.txt 1.txt 2020-01 2020-04 2022-01 2022-04 2.txt 3.txt 4.txt 5.txt 6.txt 7.txt 8.txt 9.txt new file.txt
test_file_1.txt test_file_2.txt
syedaf@MintVM:~/test_dir$ echo "${var1}"
*
```

If the variable doesn’t include wildcards the result is the same:

```
syedaf@MintVM:~/test_dir$ var1="1.txt"
syedaf@MintVM:~/test_dir$ echo ${var1}
1.txt
syedaf@MintVM:~/test_dir$ echo "${var1}"
1.txt
```

Variables...

- It is preferable to use braces - {} - to enclose a variable in order to separate it from adjacent text or variables. In the example below if you do not include the braces the variable is treated as a result of the concatenation with the adjacent string

```
syedaf@MintVM:~/test_dir$ var="Hello"

syedaf@MintVM:~/test_dir$ echo $varWorld

syedaf@MintVM:~/test_dir$ echo ${var}World
HelloWorld
```

Basic Demo Script

★ Download test csv input file from https://extendsclass.com/csv-generator.html

- Demo script to demonstrate use of the following:
    - Use of command line parameters
    - Piping command output into a variable
    - Calling a child script
        - Exporting variables to be picked up in the child script
    - Scope of Variables - Local & Global
    - Exit Codes
    - Comments

Basic Shell Script…

- Command-line parameters are ‘positional parameters’. The command itself is $0. The max number of command line parameters that can be passed in is 9 and these can be accessed using $1 through $9

- Shift is used to used to shift the position of the parameters left. This is useful when there are more than 9 parameters.

```
	syedaf@MintVM:~$ set parm1 parm2 parm3 parm4
syedaf@MintVM:~$ echo $1
parm1
syedaf@MintVM:~$ shift 2
syedaf@MintVM:~$ echo $1
parm3
```

Basic Shell Script...

- Command piping is used to chain a sequence of commands. For eg:

```
syedaf@MintVM:~/test_scripts$ cat test.csv | head -10 | tail -5
```

.. pipes the output of the file test.csv and first reads the top 10 lines. These are then piped to the
tail command and the last 5 lines are read

- Child scripts

A script can be called within another script. The calling script is called the parent script and the
called script is the child script

Basic Shell Script...

- Variable scope

  All variables declared within a function are global - accessible within the whole body of the script even though they may be defined within a function. They can be made local within a function by specifying the local keyword

- Exit codes - Every command executed by a Shell script has a return code. We get the value of the exit code from the ‘$?’ variable. An exit status of 0 means success whereas a non-zero return code means failure

- Comments - Comments are marked by beginning the line with a #. Multi-line comments can be declared by using the HereDoc:

```
      << 'MULTILINE-COMMENT'
      Line 1
                  Line 2
MULTILINE-COMMENT
```

Basic Shell Script...

- Dates are formatted using options:

```
syedaf@MintVM:~/linux_course/sed$ echo $(date)
Wed Oct 5 06:17:03 PM EDT 2022

syedaf@MintVM:~/linux_course/sed$ echo $(date +%m-%d-%Y)
10-05-2022
```

- A function is a set of encapsulated commands. Arguments can be passed in to a function similar to the manner in which they are passed in to a shell script

```
function test_function() {
        echo “arg 1: “ $1
        echo “arg 2: “ $2
}

test_function $1 $2
```

Logical Operators

- **Logical Operators are used to test conditions and can further test complex expressions by combining multiple conditions. The 3 main Logical Operators are:**

1. **&& (AND Operator) - Combines 2 conditions and the overall expression is true only if both conditions are true**

```
var1=10; var2=”Test”;
[[ $var -eq 10 && $var2 == “Test” ]] && echo true
```

2. **|| (OR Operator) - Combines 2 conditions and the overall expression is true only if at least 1 condition is true**

```
[[ $var1 -eq 20 || $var2 == “Test” ]] && echo true
```

3. **! (NOT Operator) - Returns true if the condition is false and vice -versa**

```
[[ ! -f test1.csv ]] && echo "File doesn't exist"
```

Braces, Brackets & Parentheses

- **Single Parentheses - The commands are run within a subshell. All of the commands contained are run and a single exit code is returned**

```
syedaf@MintVM:~/test_scripts$ var="Hello"

syedaf@MintVM:~/test_scripts$ echo "Var outside: $var"
Var outside: Hello

syedaf@MintVM:~/test_scripts$ ( var="World"; echo "Var in subshell: $var" )
Var in subshell: World

syedaf@MintVM:~/test_scripts$ echo "Var outside: $var"
Var outside: Hello

syedaf@MintVM:~/test_scripts$
```

Braces, Brackets & Parentheses

- **Double Parentheses - Used to perform Mathematical operations when there is no return value. The value of the variable is modified within the double parentheses. If the result within is non-zero an exit code of 0 is returned, else a return code of 1 is returned.**

```
        syedaf@MintVM:~/test_scripts$ i=1
syedaf@MintVM:~/test_scripts$ (( i+=9 ))
syedaf@MintVM:~/test_scripts$ echo $i
10
syedaf@MintVM:~/test_scripts$ (( i=3 ))
syedaf@MintVM:~/test_scripts$ echo $?
0
syedaf@MintVM:~/test_scripts$ (( i=0 ))
syedaf@MintVM:~/test_scripts$ echo $?
1
syedaf@MintVM:~/test_scripts$
```

Braces, Brackets & Parentheses

- **Dollar ($) Single Parentheses - The command inside is executed in a subshell and the value is utilized within the containing string**

```
            syedaf@MintVM:~/test_scripts$ echo "The full path of current dir is: $(pwd)"
The full path of current dir is: /home/syedaf/test_scripts

syedaf@MintVM:~/test_scripts$ var=$( inner_var=10; echo $inner_var )

syedaf@MintVM:~/test_scripts$ echo $var
10

syedaf@MintVM:~/test_scripts$
```

Braces, Brackets & Parentheses

- Dollar ($) Double Parentheses - This is similar to ((…)) except that a value is returned to the container string

```
syedaf@MintVM:~/test_scripts$ echo "The total is $(( 4 + 4 ))"
The total is 8

syedaf@MintVM:~/test_scripts$ var=$((10/2))
syedaf@MintVM:~/test_scripts$ echo $var
5
```

Braces, Brackets & Parentheses

- **Single Square Brackets - These are used to evaluate an expression to true or false. These are mainly used in checking for file/directory existence or comparing strings and integers.**

```
syedaf@MintVM:~/test_scripts$ [ -f test.sh ] && echo true
true
syedaf@MintVM:~/test_scripts$ a=10
syedaf@MintVM:~/test_scripts$ b=20
syedaf@MintVM:~/test_scripts$ [ $a -eq $b ] && echo true
syedaf@MintVM:~/test_scripts$ [ $a -ne $b ] && echo true
true
```

A complete listing of all the file related checks can be found here: https://tldp.org/LDP/abs/html/fto.html

A complete listing of string and integer related testing can be found here: https://tldp.org/LDP/abs/html/comparison-ops.html

Braces, Brackets & Parentheses

- **Double Square Brackets - These are used to evaluate an expression to true or false just like Single Square Brackets with the following differences:**

1. **Double square brackets prevent word splitting**

```
syedaf@MintVM:~/test_scripts$ var="Hello World"
syedaf@MintVM:~/test_scripts$ [ $var = $var ] && echo true
bash: [: too many arguments
syedaf@MintVM:~/test_scripts$ [[ $var = $var ]] && echo true
True
```

2. **Double square brackets allow wildcard expansion**

```
                    syedaf@MintVM:~/test_scripts$ [ Hello == H* ] && echo true
syedaf@MintVM:~/test_scripts$
syedaf@MintVM:~/test_scripts$ [[ Hello == H* ]] && echo true
true
```

Braces, Brackets & Parentheses

- Braces - These are used for expansion and ranges.

```
        syedaf@MintVM:~/test_scripts$ echo This is test# {1,2,3}
This is test# 1 2 3

syedaf@MintVM:~/test_scripts$ echo {0..5}
0 1 2 3 4 5
```

- Dollar Braces - These are used for variable interpolation to separate the expanded variable from adjacent text

```
        syedaf@MintVM:~/test_scripts$ var="Test#"
syedaf@MintVM:~/test_scripts$ echo $var1

syedaf@MintVM:~/test_scripts$ echo ${var}1
Test#1
syedaf@MintVM:~/test_scripts$
```

Std. Input, Std. Output, Std. Error

- The input or output of any Unix process is considered a file. The file descriptors are:

  0 - STDIN - Reads from our Input file (Keyboard file)

  1 - STDOUT - Writes to our output file (Screen)

  2 - STDERR - Writes to our Error file which is redirected to our Output file (Screen)

```
syedaf@MintVM:~/test_scripts$ ls
test_child.sh  test.csv  test.out  test.sh
```

In the example above ‘ls’ typed in at the prompt is Std. Input and the files listed by the command are Std. Output

```
syedaf@MintVM:~/test_scripts$ ls x.sh
ls: cannot access 'x.sh': No such file or directory
```

Std. Input, Std. Output, Std. Error

Std. Error redirection:

```
syedaf@MintVM:~/test_scripts$ ls -l test.sh x.sh 2>/tmp/err.log
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh

syedaf@MintVM:~/test_scripts$ view /tmp/err.log

syedaf@MintVM:~/test_scripts$ cat /tmp/err.log
ls: cannot access 'x.sh': No such file or directory
```

Std. Output redirection:

```
syedaf@MintVM:~/test_scripts$ ls -l test.sh x.sh 1>/tmp/output.log 2>/tmp/err.log

syedaf@MintVM:~/test_scripts$ cat /tmp/output.log
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh

syedaf@MintVM:~/test_scripts$ cat /tmp/err.log
ls: cannot access 'x.sh': No such file or directory
```

Std. Input, Std. Output, Std. Error...

The default redirection is to the output so we can achieve the same by skipping the 1 below:

```
syedaf@MintVM:~/test_scripts$ ls -l test.sh x.sh 1>/tmp/output.log 2>/tmp/err.log

syedaf@MintVM:~/test_scripts$ cat /tmp/output.log
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh

syedaf@MintVM:~/test_scripts$ cat /tmp/err.log
ls: cannot access 'x.sh': No such file or directory

syedaf@MintVM:~/test_scripts$ ls -l test.sh x.sh >/tmp/output.log 2>/tmp/err.log

syedaf@MintVM:~/test_scripts$ cat /tmp/output.log
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh
```

Std. Input, Std. Output, Std. Error...

- The Std. Error can be redirected to the same location as Std. Output as below:

```
syedaf@MintVM:~/test_scripts$ ls -l test.sh x.sh 1> /tmp/output.log 2>&1

syedaf@MintVM:~/test_scripts$ cat /tmp/output.log
ls: cannot access 'x.sh': No such file or directory
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh
```

A shortcut for redirecting Std. Output and Std. Error to the same location is:

```
syedaf@MintVM:~/test_scripts$ ls -l test.sh x.sh &> /tmp/output.log

syedaf@MintVM:~/test_scripts$ cat /tmp/output.log
ls: cannot access 'x.sh': No such file or directory
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh
```

Std. Input, Std. Output, Std. Error...

- The Std. Input can also be redirected as below:

```
syedaf@MintVM:~/test_scripts$ cat 0< /tmp/output.log
ls: cannot access 'x.sh': No such file or directory
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh
```

  Since the default for Std. Input is 0 it can be skipped:

```
syedaf@MintVM:~/test_scripts$ cat < /tmp/output.log
ls: cannot access 'x.sh': No such file or directory
-rwxr--r-- 1 syedaf test_main_group 1742 Aug 14 11:53 test.sh
```

Core Processing - Iterations

- **Looping is processed using 3 commands:**
  - **while loop**
  - **for loop**
  - **until loop**

Core Processing - Iterations

- while loop

```
while [ condition ]
do
  <list of conditions>
done
```

  - break - exits the loop when a condition is hit
  - continue - skips processing for the rest of the loop

- while loops are commonly used for file processing:

```
cat ${fileName} | while read -r line
do
            echo $line
done
```

Core Processing - Iterations

- for loop
  - for variable in (<numerical arrays> / <string arrays>)
    ```
    do
            <processing>
    done
    ```
  - for ((initializer; condition; step))
    ```
    do
        <processing>
    done
    ```

Core Processing - Iterations

- until loop

```
until [ ! condition ]
do
   <list of conditions>
done
```

  - break - exits the loop when a condition is hit
  - continue - skips processing for the rest of the loop

- Difference between the while loop and until loop: The while loop processes while the condition is true. The until loop processes until the condition is not true

IF Statements

- The IF statement is a condition statement with the following syntax:

```
          if [[ test condition ]]
then
     <commands to execute>
elif [[ test condition ]]
then
     <commands to execute>
else
     <commands to execute>
fi
```

IF Statements...

- The IF statement can be used with either:
  - Single Parentheses - ( )
  - Double Parentheses - (( ))
  - Single Brackets - [ ]
  - Double Brackets - [[ ]]

IF Statements...

- **Single Parentheses**

  These are used to evaluate a set of conditions within a sub-shell. If the set of conditions executes successfully then the commands in the IF clause are executed.

- **Double Parentheses**

  These are used to evaluate numerical conditions

- **Single / Double Brackets**

  These are used to compare strings and numbers. The difference between the two is that the double brackets has more enhanced features and allows for numeric operators in comparison and also has additional features such as wildcards

CASE Statement

- The CASE statement is used instead of the IF statement when multiple branches depend on the value of a single variable. The syntax is:

```
case <expression> in
            <pattern_1>)
            <statements>
      ;;
      <expression> in
            <pattern_x>)
            <statements>
      ;;
      *)
            default condition
      ;;
esac
```

Regular Expressions

- A Regular Expression is a sequence of characters that defines a search pattern in text
- The Regular Expression is mostly used in conjunction with Unix utilities like SED (Stream Editor) to perform operations such as search and replace
- It is handy to keep a cheat sheet for reference when working out the patterns:

  Regular Expressions Cheat Sheet by Dave Child - Download free from Cheatography.com

  Tip: Test out your regular expression at an online regex tester site to confirm your results before using them in code

  https://regex101.com/

Regular Expressions

Common patterns to check out on test file (regex.txt):

grep "a.c" regex.txt - ‘.’ is a single wildcard character in any position

grep "a.*c" regex.txt - ‘.*’ zero or more occurrences of a wildcard character in any position

grep "x\s*y" regex.txt - ‘.*’ zero or more occurrences of a space in any position

grep [abAB] test.csv - Character class. One or more of the chars between square brackets

grep [h-kA-C] regex.txt - Check for Character ranges

grep '\[.*]' regex.txt - Escape special characters such as ‘[‘ with a backslash

grep '^a.*c$' regex.txt - String beginning with a and ending with c

Regular Expressions

- **Common patterns to check out on test file (regex.txt):**

  grep -E '^[0-9]{2}$' regex.txt - Check for occurrences of 2 digits of a number

  grep -E '[0-9]{4,}' regex.txt - Check for occurrences of 4 digits or more of a number

  grep -Eo '[0-9]+' regex.txt - Check for multiple occurrences of a number

Arrays

- Bash supports both Indexed and Associative arrays. Indexed arrays have a numeric key whereas Associative arrays have a numeric or string key

- Indexed arrays are declared as below:

```
        declare -a arr=( 10 20 30 40 )
```

… or

```
        declare -a arr
        arr=( 10 20 30 40 )
```

… or

```
                arr=()
                arr[0]=10   arr[1]=20   arr[3]=30   arr[4]=40
```

Arrays

- Arrays can be looped over using either: ${arr[@]} or ${arr[*]}
- In case of string elements with spaces the array expansion should be enclosed in double spaces:

  “${arr[@]}” or “${arr[*]} “

  However, when using ‘*’ the entire array is treated as a single element and so @ is normally used

  For eg: (“One” “Two” “Three Four”) is split into 3 elements using @:

  “One” “Two” and “Three Four”

  …whereas * processes the entire element as one string: “One Two Three Four”

Arrays

- The entire array can be accessed using: “${arr[@]} “
- To access an indexed element: ${arr[0]}
- To set a particular element: arr[0]=”Value”
- To unset a particular element: unset arr[0]
- To access all the indexes: ${!arr[@]}
- To append a value to the array without using an index: arr+=(new_val)
- To find the length of an array: ${#arr[@]}
- To get a subset of an array: ${arr[@]:<start_index>:<end_index>}

AWK

- Awk is a scripting language used primarily for data extraction based on patterns and generating formatted reports from raw data

- The basic syntax of an AWK program is: (pattern) {action to be performed on pattern match}

  The default pattern matches every line if (pattern) is skipped
  The default action is to print the line if the action is skipped

- Two optional patterns are:

```
    BEGIN { actions to perform }
(pattern) {action to be performed on pattern match}
END   { actions to perform }
```

  Online reference sites:

  https://awk.js.org/ - Test AWK online
  https://quickref.me/awk - AWK reference

AWK

- Awk can be run in 3 formats:

  1. From the command line as a standalone statement

     awk -v <variable_list> <rest of awk statements>

  2. Inside a bash script

  3. From the command line but inside an awk file

     awk -f <awk_file> <input_file>

- The GNU Awk reference is available at:

  https://www.gnu.org/software/gawk/manual/gawk.html

- Awk Cheatsheet:

  Awk Command Cheat Sheet & Quick Reference (https://quickref.me/awk)

Common Awk Variables

- ARGC - # of arguments at command line - 1 (the word ‘awk’ is considered one argument)
- ARGV - array storing the command line arguments
- FNR - number of records in file
- FS - field separator
- NF - number of fields
- NR - total number of records
- OFS - output field separator
- ORS - output record separator
- RS - input record separator

Awk Regular Expressions

- Regular Expressions are used in Awk to search for and replace patterns in the input file:

  For matching a pattern:

  ```
  awk -e '<word/s> ~ /<pattern to match>/ {action}' file
  ```

  For NOT matching a pattern the exclamation mark is used:

  ```
  awk -e '<word/s> !~ /<pattern to match>/ {action}' file
  ```

  For replacing all occurrences of the pattern:

  ```
  awk '{gsub(/{SEARCH_PATTERN}/,{REPLACE_PATTERN}); print}' {file}
  ```

SED

- SED is an abbreviation for Stream Editor. It is mainly used for searching and replacing strings in files.

  The syntax is:

  sed [options] <input_file>

  Common examples of usage:

  - sed ‘s/<search>/<replace>/’ filename
    – replaces the search pattern with the replace pattern. Only the first occurrence is replaced.

  - sed ‘s/<search>/<replace>/g’ filename
    – replaces every occurrence in the line

  - sed ‘s/<search>/<replace>/n’ filename
    – replaces the nth occurrence in the line

SED

- sed ‘1,ns/<search>/<replace>/g’ filename
  – replaces the word in a range of lines in the file
- sed ‘s/<search>/<replace>/n’ filename
  – replaces the nth occurrence in the file
- sed ‘nd’ filename
  – deletes the nth line in the file
- sed ‘n1,n2d’ filename
  – deletes the range of lines in the file
- sed '/<pattern>/d' filename
  – deletes a pattern in the file

Grep

- Grep searches a file for a pattern of characters and returns the lines that match. Some of the common options are:

grep -i <pattern> filename - - Case insensitive

grep -n <pattern> filename - - Returns the line number too

grep <^pattern> filename - - Return lines that begin with the pattern

grep <pattern$> filename - - Return lines that end with the pattern

grep -r <pattern> - - Recursive grep through sub-directories

Grep

- Grep can be used within a shell script in the following cases:
  - Check for a true/false condition

```
if grep -q <pattern> filename; then
        <success>
else
        <failure>
fi
```

  - Loop through multiple lines from the result

```
                grep <pattern> filename | while read a;
do
        …
done
```

Useful Utilities

- **The cut command is used to extract columns from piped inputs or files based on delimiters**

  Cut columns 1 and 2 from a csv file: cut -d',' -f 1,2 <filename>

- **The uniq command is used in conjunction with the sort command to eliminate duplicate lines**

  Sort then get unique lines: sort -t '<delimiter>' -n (numerical) -k<col#> <input_file> | uniq

Scheduling Jobs - Cron

- **The Cron utility is used to schedule jobs to run at fixed intervals. Crontab is short form for ‘Cron table’ and is a table used to list all the scheduled Cron jobs.**

- **The common crontab commands are:**

  List the crontab for the user - ‘crontab -l’

  Edit the crontab - ‘crontab -e’   - Gives an initial choice of an editor

  Remove the crontab - ‘crontab -r’

  Display the last time edited - ‘crontab -v’

Scheduling Jobs - Cron

- A starter template for the crontab is listed below:

```
# .---------------- Minute (0 - 59)
# |  .------------- Hour (0 - 23)
# |  |  .---------- Day Of Month (1 - 31)
# |  |  |  .------- Month (1 - 12) OR Jan,Feb,Mar,Apr ...
# |  |  |  |  .---- Day Of Week (0 - 6) (Sunday=0 OR 7)  OR Sun,Mon,Tue,Wed,Thu,Fri,Sat
# |  |  |  |  |
# *  *  *  *  *  Command to be executed
```

Cheat Sheet: https://quickref.me/cron.html

Processing Input

Script Input is processed using the following options:

- Using command-line arguments to the script
- Read arguments from a prompt

    read - Read data from the keyboard

    read -p - Read data from a prompt

    read -s - Read typed data in silent mode (eg. passwords)

    $REPLY - Read the whole line of input data. There is no splitting into multiple arguments

- Read a file line-by-line in a loop
- Process the parameters using ‘getopts’

Processing Input...

Getopts is a built-in command of the bash shell. It processes command options and arguments. It is the bash version of the system tool - getopt.

- **getopts processes short options which is a ‘-’ followed by a letter or digit**
- **The positional parameters are stored in the shell variable ‘$@’**

  For the input: ‘command -x value_1 -y value_2’ the value of the variable ‘$@’ is ‘-x value_1 -y value_2’

- **getopts is processed by running it in a loop (while) and processing each option in a loop. The loop is terminated when there are no more options to process.**
- **For example we will discuss the following line processing getOpts in our demo script:**

Processing Input…

- while getopts ":a:bc:" optName; do
  … <rest of while loop>
- “abc” is the option string with the 3 options being a, b and c
- The colon preceding a specifies that getopts does not report errors but the script has to process the errors itself
- The colon following ‘a’ specifies that option ‘a’ needs an argument
- There is no colon following b so it does not require an argument
- The colon following ‘c’ specifies that option ‘c’ needs an argument
- The optional arguments begin after all the options and their arguments have been specified
- Given the options above, this is what our command line would look like:

```
<scriptName> -a <argA> -bc <argC> <inputFileName>
```

Processing Input…

- **For <scriptName> -a <argA> -bc <argC> <inputFileName>**
  …and the script snippet below:

- **while getopts ":a:bc:" optName; do …**

- **The first run of the while loop will place ‘a’ in optName and <argA> in $OPTARG. The second run will place ‘b’ in optName but there is no $OPTARG since there is no following colon. The third run will place ‘c’ in optName and <argC> in $OPTARG. The ‘-’ preceding ‘a, b and c’ indicates that they are options**

- **If the option does not match ‘a,b or c’ optName is set to ‘?’. If the option matches but there is no argument then optName is set to ‘:’**

  <inputFileName> follows all the options and arguments. It is the optional argument.
  ${1} is ‘-a’, {2} is <argA>, {3} is ‘-bc’, {4} is <argC>, {5} is <inputFileName>

  shift "$((OPTIND - 1))" …. resets {1} to the beginning of the optional arguments which is <inputFileName> and any arguments after it would be assigned {2}, {3} etc

Processing Output…

- Output to stdout:

```
echo <text to display>
```

- Output can be piped out to a file using redirection

```
echo “<text to output>” > ${outputFile} – (> creates a new file)
echo “<text to output>” >> ${outputFile} – (>> appends to an existing file)
```

- Display in stdout as well as file:

```
echo “<text to output>” | tee ${outputFile}
echo “<text to output>” | tee -a ${outputFile} — (Append)
```

- Send stdout and stderr to the same file:

```
echo “<text to output>” >> ${outputFile}
<Invalid command> >> ${outputFile} 2>&1
```