Linux Notes

Dr. J. V. Smart

Syllabus

Syllabus with effect from the Academic Year 2021-2022

Course Code	PS01CMCA54	Title of the Course	OPERATING SYSTEMS
Total Credits of the Course	4	Hours per Week	4

Course Objectives:	1. To provide basic understanding of the role and functioning of an operating system. 2. To introduce Linux shell environment and programming.

Course Content

Unit	Description	Weightage* (%)
1.	**Introduction to Operating Systems** - Understanding the role of operating systems - Operating system services - Operating system structure - The concepts of interrupt handling, system call, shell, operating system interface	25

Unit	Description	Weightage* (%)
	- Virtual machines - Linux Bash shell programming fundamentals - Command-line processing - Bash shell variables, control structures - input, output, integer arithmetic, string operations	
2.	**Process Management** - The concept of a process - Scheduling of processes - Interprocess communication - Multithreading: concepts, advantages, models - Schedulers: long term, middle term, short term - CPU scheduling: criteria and algorithms - Multiprocessor scheduling - Introduction to process synchronization - The critical section problem and Peterson's solution - The concepts of semaphores and monitors - Introduction to deadlocks	25
3.	**Memory Management and File Systems** - Basic concepts of memory management - Paging - Segmentation - Virtual memory, demand paging - Page replacement - Introduction to file system management and directory structure - File system mounting - Disk scheduling	25

Unit	Description	Weightage* (%)
4.	**Linux Shell Programming - The vim editor - File system manipulation commands - I/O redirection - Regular expressions - Basic filters - The sed and awk commands	25

Teaching-Learning Methodology	Blended learning approach incorporating traditional classroom teaching as well as online / ICT-based teaching practices

Evaluation Pattern

Sr. No.	Details of the Evaluation	Weightage
1.	Internal Written / Practical Examination (As per CBCS R.6.8.3)	15%
2.	Internal Continuous Assessment in the form of Practical, Viva-voce, Quizzes, Seminars, Assignments, Attendance (As per CBCS R.6.8.3)	15%
3.	University Examination	70%

Course Outcomes: Having completed this course, the learner will be able to	
1.	describe the role and functioning of an operating system.
2.	demonstrate understanding of fundamental concepts related to operating systems.
3.	understand process, memory and file system management.
4.	gain familiarity with Linux command line environment.
5.	use basic Linux commands.
6.	develop Linux shell scripts.

Suggested References:

Sr. No.	References
1.	Silbetschatz, Galvin, Gagne: Operating System Concepts, 8th edition, John Wiley and Sons, Inc., 2008
2.	Kochan S. G., Wood, P. : Unix Shell Programming, 4th edition, Addison Wesley, 2016
3.	Das S. : UNIX and Shell Programming, Tata McGraw-Hill Education, 2008
4.	Nutt G. : “Operating Systems” : 3rd Edition, Pearson Education, 2004
5.	Tanenbaum A. S., Woodhull A.S. : “Operating Systems Design and Implementation”, 3rd edition, Prentice Hall, 2006
6.	Shotts W. : “The Linux Command Line: A Complete Introduction Illustrated Edition”, 2nd Edition, No Starch Press, 2019

Glossary

Kernel

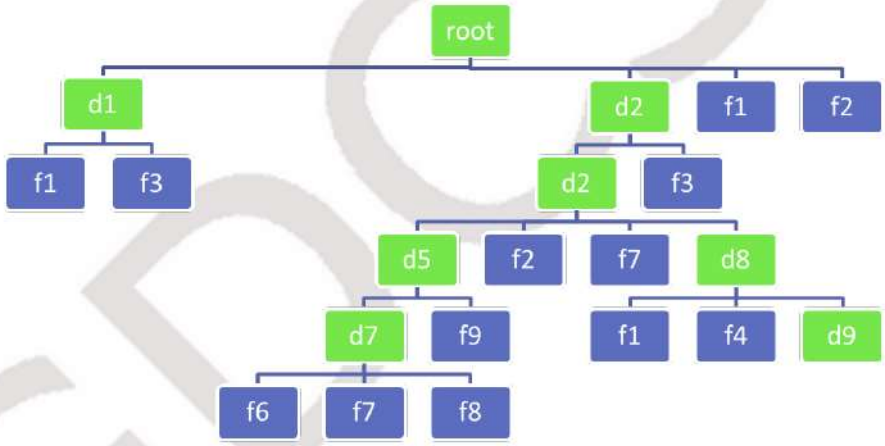
The operating system kernel is the one program running on the computer system at all times. The kernel provides the interface between the hardware and the programs

Shell

A shell is an interface between the user and the operating system. A shell provides either a Graphical User Interface (GUI) or a Command Line Interface (CLI)

Introduction

The Unix/Linux File System



The file system structure

- Hierarchical file system
  - While the Windows file systems is actually a forest, the UNIX file systems is a proper tree (actually a directed acyclic graph)
- The root directory is identified by the / (slash) character
- Directories and files
- Identifying a file system object uniquely using path
- Components of the path are separated by the / (forward slash) character
- / (forward slash) v/s \ (backslash)
- Filenames are case sensitive
- Extensions not an essential part of file name (executable files usually have no extension)
- Any character except / can be used in file names
- The notion of current directory and relative paths
- Absolute path v/s relative path
- Interactive users have a home directory. A user has full permissions on one's home directory. The ~ (tilde sign) is a shortcut for the user's home directory in file system manipulation commands
- Hidden files and directories - any file or directory whose name starts with a . is considered hidden. Hidden files and directories are not displayed in the nautilus file browser (GUI) or by the ls command (CUI). However, there are options to display them

Accessing Multiple File Systems

A computer system may have many storage devices in it. Also, removable devices may be inserted or attached and removed at any time. Each device has

its own file system on it. A device that can have multiple partitions, like the hard disk, has a separate file system on each partition. How does one access these file systems? Operating systems like Microsoft Windows assign a separate drive letter (like C:, D:, E:, etc.) to each file system. However, Linux and other Unix-like systems have a single file system tree starting with the root directory, denoted by / (the slash character). The file system contained on the partition from which Ubuntu boots is called the root file system. The root directory of this file system becomes / - the root of the entire file system tree. Initially this is the only file system available.

We may access any other file system by mounting it on any existing directory (this directory is called the mount point). Once mounted, the contents of that file system appear as the contents of the mount point directory. If the mount point previously contained some contents (files and subdirectories), they are masked (hidden) for the duration of the mount. Now we may access (and modify) the contents of that file system from the mount point directory. When we no longer need to use the file system, we may unmount it. At this point, the original contents of the mount point directory get unmasked (become visible again). This process is depicted in the following figures a, b, c and d. Figure a shows the root file system. Figure b shows the file system on another device. Figure c shows the situation after mounting the file system of figure b onto the directory d3 of Figure a. The original contents of d3 are now masked and the contents of the file system mounted there appear as if they are the contents of d3. Figure d shows the situation after unmounting the second file system. The original contents of the directory d3 now become visible again.

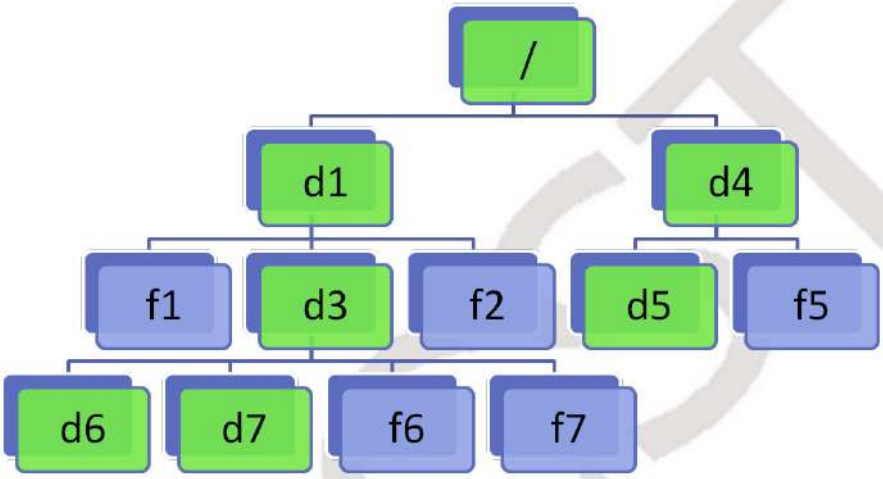


Figure a: The root file system

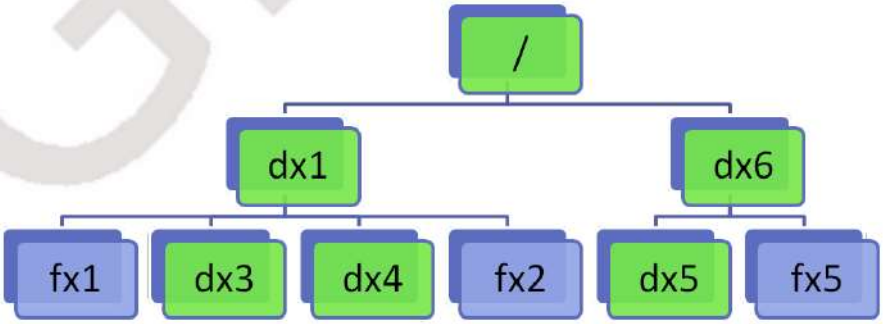


Figure b: File system on another device

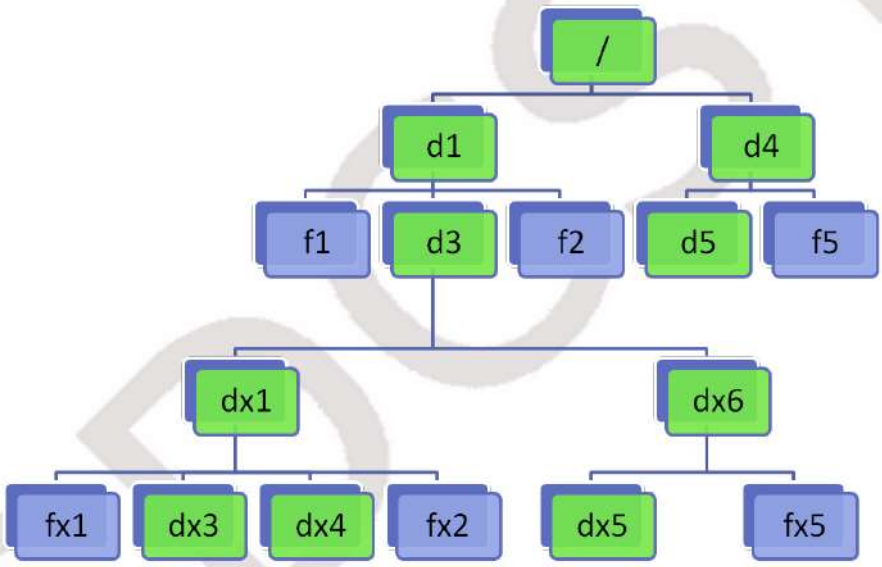


Figure c: After mounting the second file system on directory d3

Figure d: After unmounting the second file system from directory d3

Figure d: After unmounting the second file system from directory d3

However, the common practice is to mount file systems onto empty directories. In the default configuration, Ubuntu automatically detects other fixed devices in the system and shows them in the places menu and the left pane of the file browser. They are mounted when we first try to access them. The file browser shows a triangular icon alongside all mounted file systems other than the root file system. The file system can be unmounted by clicking on this icon. It may be mounted again when the user tries to access it again next time. The root file system cannot be unmounted. Removable devices are automatically mounted when they are inserted. Read-only media like optical disks can be unmounted

by simply removing or ejecting them. Media on which writing is possible (like USB flash disks) must be unmounted by clicking the unmount icon in the file browser or by right clicking its icon on the desktop and selecting “safely remove device” option. This causes any data cached in main memory for improved performance to be flushed to the disk. A message at the end of this process announces that it is now safe to remove the device and the unmount icon disappears from besides the device’s entry in the file browser’s left pane. Only after this the device can be safely detached from the system. Failure to observe this procedure may result in loss of data or damage to the file system on the removable media. While this method of accessing the other storage devices in the system may sound unnecessarily complicated, it is more flexible and powerful and has several distinct advantages.

Not only local file systems, even the remote file systems can also be accessed in the same way. For example, Windows shares can be mounted on local directories using the SMB (Server Message Block) protocol and then be accessed just like local content. Similarly, NFS (Network File System), FTP server directories can also be mounted on a directory and accessed as a part of the file system.

Of course, all these procedures can also be carried out using commands. It is also possible to configure the system to mount some file systems at particular mount points automatically every time the system boots. By default, the system mounts other fixed devices in the system under the `/mnt` directory and removable devices in the system in directories under the `/media` directory.

The Shell

- The shell acts as an interface between the user and the operating system
- The shell provides the user interface through which the users can start and stop programs
- The shell can be text-mode (Command Line Interface - CLI) or graphical mode (Graphical User Interface - GUI)
- File Explorer / Windows Explorer is the graphical shell that comes with Microsoft Windows
- cmd is the text-mode shell supplied along with Microsoft Windows
- Linux systems offer a variety of shells in both the GUI mode (GNOME Shell, KDE, Unity, XFCE, LXDE, MATE, etc.) and the text mode (sh, bash, ksh, csh, etc.)

The Linux Shells

Unix offers a variety of shells in both text mode and graphical mode. The graphical mode shells include the GNOME shell, Unity, KDE, XFCE, LXDE, MATE, etc. The commonly used text mode shells include sh (the original Bourne shell), ksh (Korn shell) csh (C Shell), bash (Bourne-Again SHell), etc.

The Bourne Shell

- **Internal commands** are built into the shell itself. There is no separate executable file for them
- **External commands** are not built into the shell. There is a separate executable file for them
- **Search path**
  - Since it is not practical to search the entire hard disk for the executable file corresponding to a possible external command entered by the user, only the directories in the search path are searched for the same. The search path is stored in the built-in shell variable PATH as string containing one or more directories separated by the : (colon) character
  - The current directory is not searched for a command if it is not in the search path
- CTRL-D at the beginning of a line indicates end-of-file (or end-of-input)
- CTRL-C is used to terminate the currently running foreground process
- CTRL-S is used to pause the terminal
- CTRL-Q is used to resume the terminal
- To copy text from the terminal, select the text using the mouse and press CTRL-SHIFT-C
- To paste text into the terminal, press CTRL-SHIFT-V. The text is inserted as if it had been typed by the user
- CTRL-L is used to redraw / refresh / clear the terminal
- The shell maintains the history of the commands entered earlier in memory. When the shell session terminates, this command history is saved in the file ~/.bash_history. The command history can be accessed using the UP / DOWN arrow keys at the prompt. It is also available in subsequent sessions and survives reboots
- The shell has its own full-fledged built-in programming language with variables, control structures, input, output, etc. It is a dynamic language in nature
- Case sensitivity

- Command completion using TAB and TAB-TAB

The Command Line

- **Components of the command line and word splitting** The command line contains one (or more) command(s), arguments to the command, operators, etc. The shell performs word splitting on the command line using white space (space / tab characters) as the separators
- In general, order of components of a command line is not important. However, there are several exceptions
- **Options**
  - **Short options** Short options consist of - (hyphen) followed by a single character. For example, `-R`
  - **Long options** Long options are specified using -- (two consecutive hyphens). For example, `--recursive`
  - **Options with arguments** Some options have their own arguments. In such cases, the argument(s) must immediately follow the option
  - **Combining short options** Multiple short options can be combined by writing them immediately after a single hyphen: `ls -l -h` can also be written as `ls -lh`. Only the last short option can have argument(s) in such case; and the argument, if any, is immediately specified after the combined options: `tar -czf backup.tar.gz directory`
- **Quoting and escaping** Strings on the command line can be specified without using any quotes. However, if a string contains white space or some other character having a special meaning to the shell interpreter, the string must be enclosed in quotes or the white spaces / special characters must be escaped
  - **Escaping** A character having special meaning to the shell can be escaped by preceding it with \ (backspace). This removes its special meaning. The special meaning of the backslash itself can also be removed by preceding it with another backslash `\\`
  - **Single Quotes** Almost no processing is carried out inside strings enclosed in single quotes `'aaa bbb ccc'`
  - **Double Quotes** Double quotes also behave like single quotes, but variable / parameter expansion *is* carried out inside double quotes: `$variable_name` is converted into the value of the variable. Hence if the value of variable *name* is *Scotty*, `echo 'Beam me up, $name'` outputs `Beam me up, $name` but `echo "Beam me up, $name"` outputs `Beam me up, Scotty`

- **Operators** (Input/Output Redirection)

File System Manipulation Commands

Note: *File system object* means either a file or a directory

- **pwd** (present working directory)
  - **pwd** Outputs the current working directory
- **cd** (change directory)
  - **cd directory** Changes the current directory to the given directory
  - **cd** changes the current directory to the home directory of the user. This behaviour is different from Windows where `cd` displays the current directory
- **ls** (list)
  - **ls** Outputs the names of the file system objects in the current directory
  - **ls arguments** Outputs the information about the argument(s). In case an argument is a file, its name is displayed. In case an argument is a directory, the names of file system objects inside the directory are displayed. If the argument file system object does not exist, an error message is output
  - **Options**
    - **-l** (*long* listing) Outputs a long listing with more information about the file system objects
    - **-d** (*directory*) When an argument is a directory, outputs information about the directory itself rather than its contents
- **mkdir** (make directory)
  - **mkdir directory** Creates the given directory if it does not exist
- **rmdir** (remove directory)
  - Removes (deletes) the given directory if it is empty
- **cat** (concatenate)
  - **cat file(s)** Outputs a concatenation of the given files. Also used to output a single file
- **cp** (copy)
  - **cp arguments** There must be at least two arguments. The last argument is the target. All earlier arguments are sources. *cp* does not copy directories by default. There are no warning or prompts if a file is going to be overwritten
    - **copy a file to another file** `cp source-file destination-file`
    - **copy one or more files to another directory** `cp file1 file2`

`file3 ... destination directory`

- **Options**
  - **-R** (recursive) This option is used to copy a directory along with all its contents recursively
  - **-i** (interactive) Prompts before overwriting a file

- **rm** (remove)
  - **rm arguments** Removes (deletes) the given arguments. While the files and directories deleted from the GUI go into the trash, files and directories deleted from the command prompt are permanently lost. `rm` does not delete directories by default. It does not prompt before deleting a file
  - **Options**
    - **-R** (recusrive) Directories in the arguments are deleted with all the contents recursively
    - **-i** (interactive) Prompts before deleting a file / directory
- **mv** (move)
  - **mv arguments** Moves or renames as per the number and types of arguments. Does not prompt before overwriting a file
    - **Rename a file or directory** `mv old-name new-name`
    - **Move a file or directory to another directory** `mv source target-directory`
  - **Options**
    - **-i** (interactive) Prompts before overwriting a file

Shell Globbing Patterns

Pattern	Matches with
?	Any single character
*	Any number of any characters

Examples

Pattern	Matches with
p*	Anything starting with p
p*e	Anything that starts with p and ends with e, including pe
*~	All backup files

Pattern	Matches with
b??h	b, followed by exactly two characters, followed by h
b*h	Anything that starts with b and ends with h, including bh
dir*	Anything that starts with dir, including dir
dir?	dir followed by exactly one character

Plain Text Editors

- Plain text editors
  - Graphical editors
    - gedit / Text Editor (default)
    - gvim
    - leafpad
  - Text mode editors
    - vi
    - vim (VI iMproved)
    - nano, pico
    - emacs (Editing MACroS)

Brief Overview of Working with the Vim Editor

- vim stands for VI iMproved
- It is an improved version of the vi editor
- vim has several *modes*
  - *Normal* (*command*) mode
    - When vim starts, it is in this mode
    - In this mode, the keys pressed by the user are interpreted as commands and corresponding actions are taken
  - *Insertion* mode
    - This mode is entered by pressing the *i* command in the command mode. Several other commands also switch to insertion mode
    - in this mode, the keys pressed by the user are treated as characters to be inserted into the text being edited at the curent cursor position
    - One may switch back to the command mode by pressing `ESC` in the insertion mode

    - *Last line* mode
        - This mode is entered when using certain commands in the command mode, like `:`
        - It is used for *ex commands* and long commands
        - The cursor moves to the last line where the rest of the command is entered
        - The command is completed (and executed) by pressing `ENTER` or `ESC`
        - To cancel the command, press `CTRL+C`
- Cursor movement keys like the arrow keys, HOME, END, PgUp, PgDn, etc. work as expected in both the command mode and the insertion mode in vim
- When vim is installed, the command *vi* also invokes *vim* only
- A file is opened by passing its name on the command line

```
vi filename
```

- A file is saved by typing `:w` (write) in the command mode
- To save the current file and close vi, type `:wq` (write and quit) in the command mode
- To close vi without saving the current file, type `:q!` (quit, with force) in the command mode
- An entire line can be deleted by pressing `dd` in the command mode
- *n* lines can be deleted by pressing `ndd` in the command mode. Here *n* is an integer and is known as the repeat count
- Deleting line(s) actually *cuts* them, so they can be pasted immediately afterwards
- `yy` (yank = copy) copies a line of text
- *n* lines can be copied by pressing `nyy` in the command mode.
- Text that is copied or cut (deleted) can be *put* (pasted) after the cursor using the command `p` (put) and before the cursor using the command `P`
- `u` (undo) can be used to undo the operations, while `CTRL-R` (redo) can be used to redo the operations that were previously undone

The nano Editor

`nano` is a very simple text editor. It can be invoked by typing `nano filename` at the prompt. Once the file is open, one can type any content one wishes. The following operations are supported:

- **CTRL+O ENTER** Save the current file
- **CTRL+X** Close the file and exit the nano editor. If the file is already saved, it will exit immediately. If the file is not saved, it will prompt `Save modified buffer?`. Press `y` for `yes`. Then it will prompt `File name to write:`. Press `ENTER` to accept the current file name. It will save the file and exit

The Shell Scripting Language of the bash Shell

Shell Variables

- The shell variables need not be declared. A variable is created automatically the first time it is assigned a value. All variables are of type *string*; but, in some contexts, may be interpreted as numbers. The variable assignment takes the form

```
variable_name=value
```

There must not be spaces around the = (assignment operator). It is not necessary to enclose the value in quotes. However, if the value contains spaces or some special characters, then you will need to use quoting or escaping. Also, there is no `$` in front of the variable name when assigning value to it.

- The value of a variable can be accessed in the following ways:

```
$variable_name
${variable_name}
```

- It is ***not*** an error to attempt to access the value of an undefined variable. Such an attempt simply returns an empty string (a string of zero length)
- **Tip** Whenever you have a doubt that a variable may not exist, or that its value may be empty string or that its value may contain spaces in it, access its value using `"` (double quotes). For example, use `test -n "$v1"` instead of `test -n $v1`. You may also choose to always use double quotes
- **Built-in variables of the shell** The shell has a number of built-in variables. By convention, their names are in ALL_CAPS. For example, PATH, HOME, SHELL, TERM, etc.

- **SHELL** : The current shell
- **HOME** : Home directory of the currently logged in user
- **PATH** : A : (colon) separated list of directories that are searched for external commands
- **TERM** : The type of the current terminal
- **PS1** : The primary prompt
- **PS2** : The secondary prompt

The `test` Command and the `[` Builtin

- The test command
  - The command `test` tests some condition and returns exit status accordingly (0 means success / true and non-zero means failure / false). It does not produce any output
  - The spacial parameter `?` can be used to output the exit status of the last foreground process (`echo $?`)
  - The syntax `[ condition ]` can be used in place of test condition (`[` is a shell builtin)
  - There **must be** at least one space between each and every argument and `[` , `]`

```
echo $?
```

- String tests
  - ***-z string*** Returns true if *string* has zero length (empty string)
  - ***-n string*** Returns true if *string* has non-zero length
  - ***string1 = string2*** Returns true if *string1* is equal to *string2*
  - ***string1 != string2*** Returns true if *string1* is not equal to *string2*
- Integer tests
  - ***no1* -gt *no2*** Returns true if *no1* is greater than *no2*
  - ***no1* -ge *no2*** Returns true if *no1* is greater than or equal to *no2*
  - ***no1* -lt *no2*** Returns true if *no1* is less than *no2*
  - ***no1* -le *no2*** Returns true if *no1* is less than or equal to *no2*
  - ***no1* -eq *no2*** Returns true if *no1* is equal to *no2*
  - ***no1* -ne *no2*** Returns true if *no1* is not equal to *no2*

The Control Structures

- The *if* statement

```
if test condition1
then
    statement1
    statement2
    ...
elif test condition2
then
    statement3
    statement4
    ...
elif test condition3
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

```
if [ condition1 ]
then
    statement1
    statement2
    ...
elif [ condition2 ]
then
    statement3
    statement4
    ...
elif [ condition3 ]
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

- The *while* statement

```
while test condition
do
    statement(s)
done
```

```
while [ condition ]
do
    statement(s)
done
```

- The *for* statement

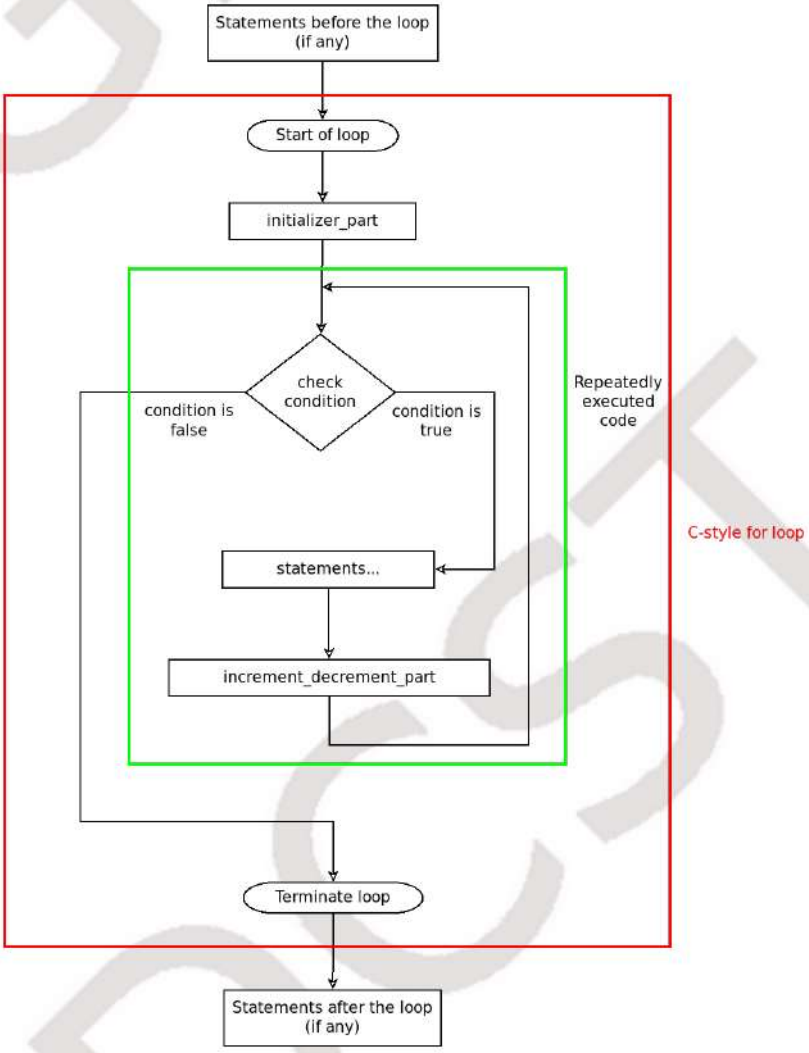
```
for variable in white-space-separated-list
do
    statement(s)
done
```

- The *C-style for* statement

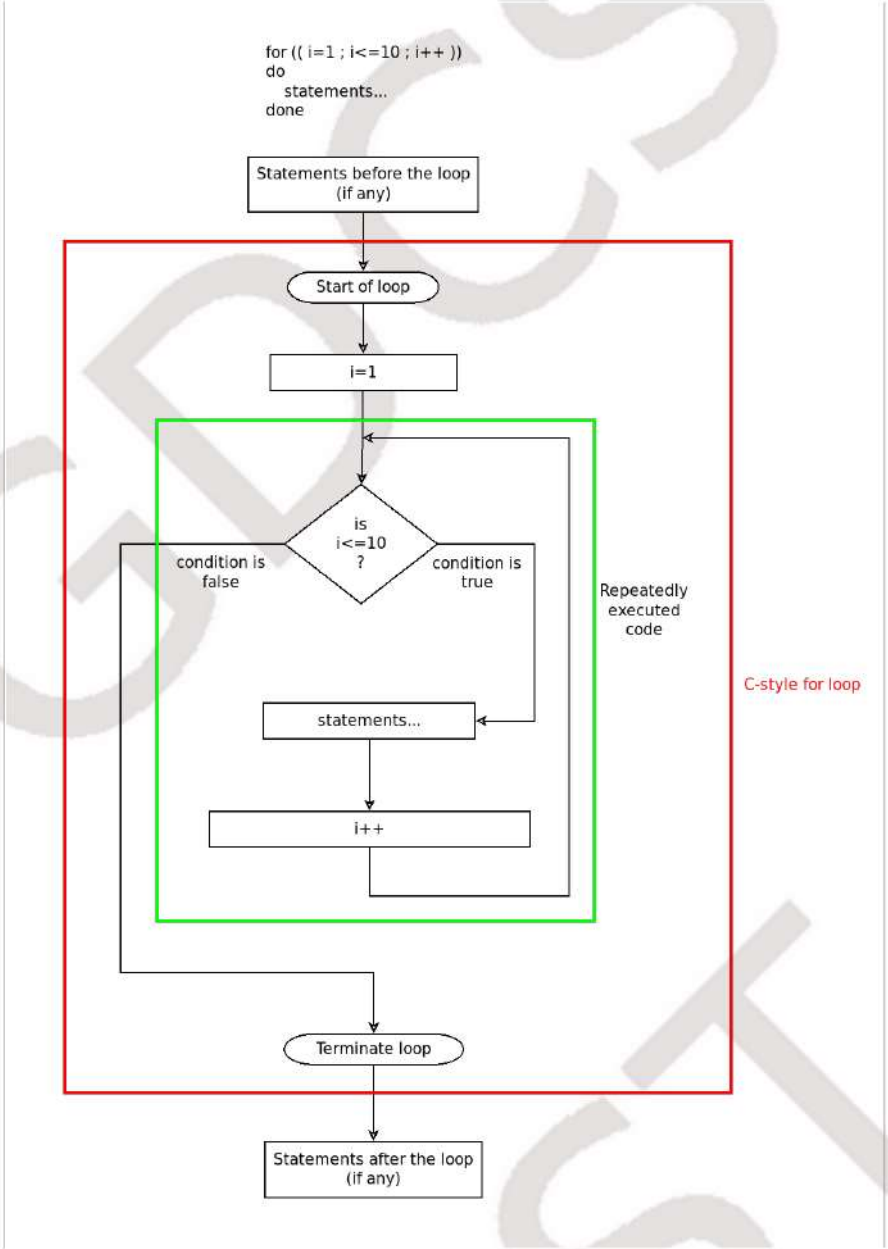
```
for ((initializer; test; increment))
do
    statement(s)
done
```

```
for ((i=1;i<=10;i++))
do
    echo $i
done
```

```
for (( initializer_part ; condition ; increment_decrement_part ))
do
   statements...
done
```



The C-style for loop



The C-style for loop example

- New line in syntax
  - A `;` (semicolon) can be used wherever the syntax requires a new line
- break and continue
  - `break` can be used in the loops to terminate the loop and jump to the next statement after the loop
  - `continue` can be used to terminate the current iteration of the loop and jump to the start of the loop
- The command `true` does nothing, but returns with an exit status of zero (success / true)
- The command `false` does nothing, but returns with a non-zero exit status (failure / false)

```
i=0
while true
```

```
do
    echo $i
    let i++
    if [ $i -ge 10 ]
    then
        break
    fi
done
```

Comments

- The **#** character is used for writing single-line comments
- All the characters starting from the # up to the end of the line are treated as comment and are ignored by the shell interpreter
- There is no syntax for multi-line comments

Output

- *echo*
  - `echo` is used to output its arguments
  - By default, echo appends a new line at the end of its output
  - To suppress that new line, use the `-n` option:

```
echo Your result is $result
echo "vaccination status: $status"
echo -n "Enter your name: "
```

Input

- *read*
  - The `read` shell builtin can be used to read the values of one or more variables
  - `read x` reads a line from the input and assigns it to the variable x
  - The `-p` option can be used to display a prompt for reading

```
read x
read -p "Enter a number: " no
```

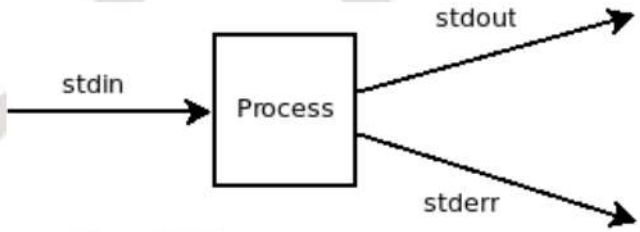
Integer Arithmetic

- *let*
  - `let` is a shell builtin for integer arithmetic
  - `let` expects an expression as a single argument (so the expression must not have spaces in it; or it should be quoted)
  - `let` supports the increment operator `++` and the decrement operator `--`. `let i++` adds `1` to the value of the variable `i`. It is same as writing `i=i+1`. `let i--` subtracts `1` from the value of the variable `i`. It is same as writing `i=i-1`.

```
let x=10*20
let x=(10+20)*20
let x=y++
let i++
```

Standard I/O Streams and I/O Redirection

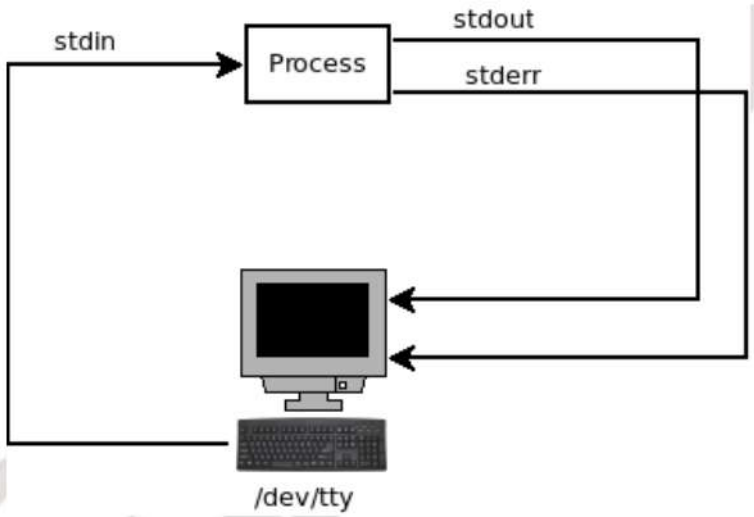
Every process has three standard streams opened by default - standard input, standard output and standard error. Standard input is used for input. By default the input comes from the keyboard. Standard output is used for normal output. Standard error is used for output of error messages. Both standard output and standard error go to the display device by default. However, these standard streams can be redirected to somewhere else using input / output redirection.



Standard Console I/O Streams

Default Assignment of Standard I/O Streams

```
$ p1
```



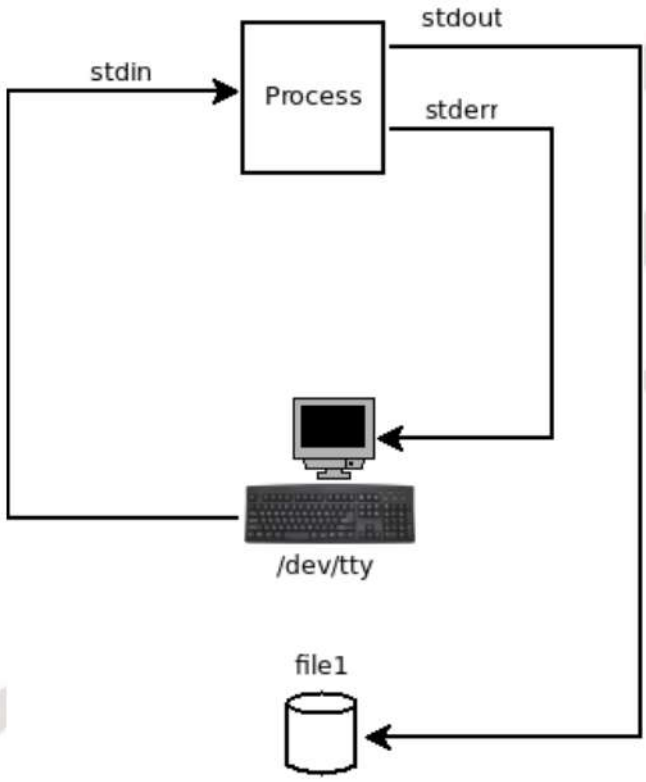
Default Assignment of Standard I/O Streams

Standard Output Redirection

```
$ p1 1>file1

$ p1 >file1

$ >file1 p1
```



Standard Output Redirection

The file *file1* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

```
$ >t1
```

This will create the file t1, if it does not exist; and will truncate the file t1, if it does.

```
$ xyz >file1     #there is no command called xyz in the path
bash: xyz: command not found
```

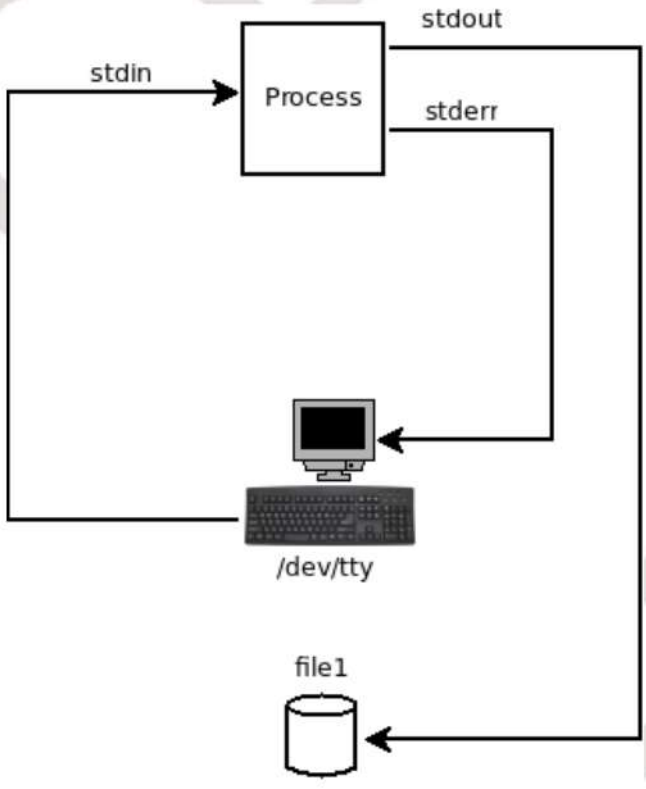
But the file file1 is still overwritten (truncated in this case).

Standard Output Redirection (Append Mode):

```
$ p1 1>>file1

$ p1 >>file1

$ >>file1 p1
```



Standard Output Redirection

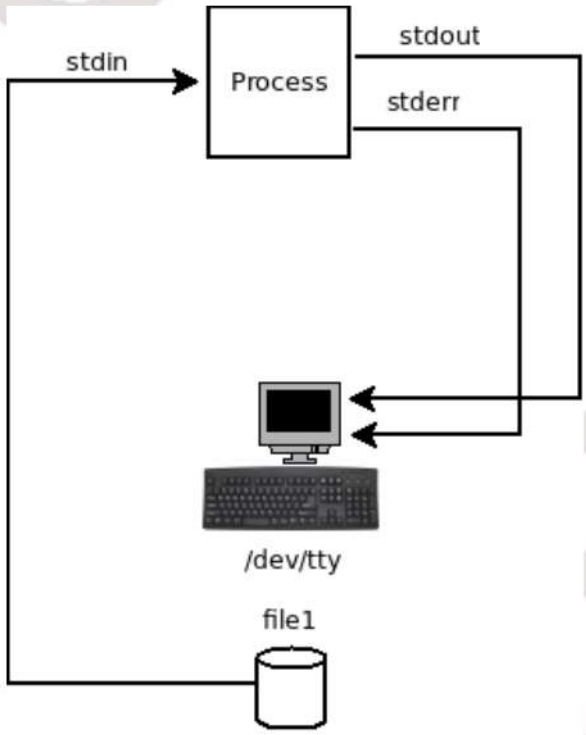
The file *file1* will be opened in *append* mode by the shell *before* execution of the command.

Standard Input Redirection

```
$ p1 0<file1

$ p1 <file1

$ <file1 p1
```



Standard Input Redirection

The file *file1* will be opened in *read* mode by the shell *before* execution of the command. If the file does not exist, it is an error.

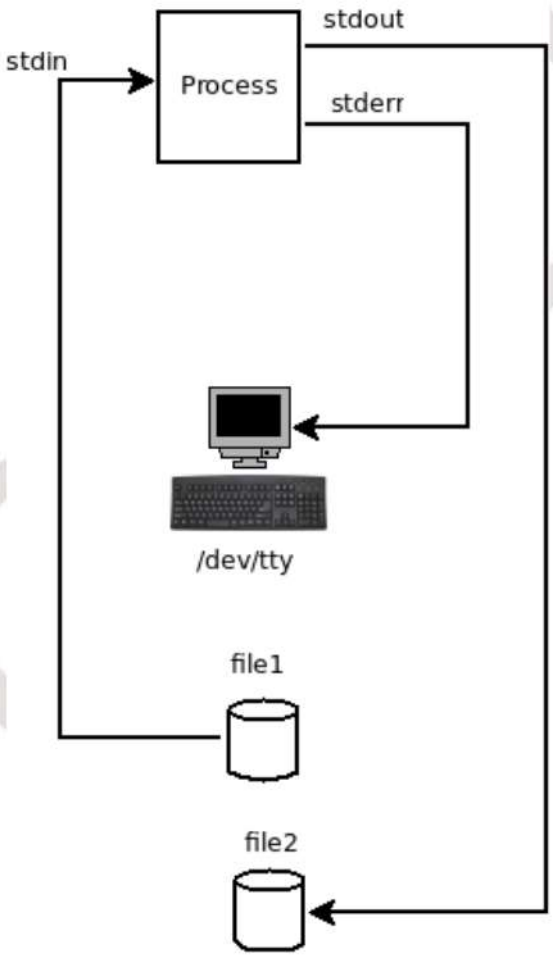
Standard Input and Standard Output Redirection

```
$ p1 0<file1 1>file2

$ p1 <file1 >file2

$ p1 >file2 <file1

$ >file2 <file1 p1
```



Standard Input and Standard Output Redirection

The file *file1* will be opened in *read* mode by the shell *before* execution of the command. If the file does not exist, it is an error.

The file *file2* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

file1 and *file2* must not be same files. If the following command is executed:

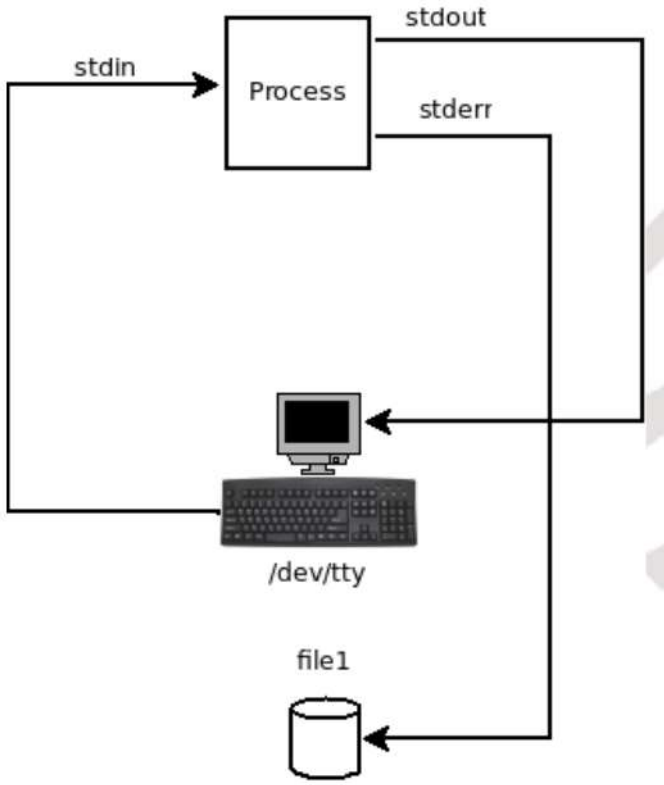
```
$ p1 <zzz >zzz
```

The file *zzz* will be overwritten *before* the execution begins and it will be an empty file.

Standard Error Redirection

```
$ p1 2>file1

$ 2>file1 p1
```



Standard Error Redirection

The file *file1* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

If we use 2>> operator in place of 2> operator, the file will be opened in append mode.

```
$ p1 2>>file1
```

/dev/null is the null device. Any output sent to it is simply discarded. If we try to perform input from it, we immediately get EOF (End-of-File).

```
$ find / -name '*.php' 2>/dev/null
```

Command Substitution

Command substitution means constructing the command line by substituting a command in it by the standard output of that command.

```
$ p1 p1_arg1 p1_arg2 `p2 p2_arg1 p2_arg2 p2_arg3 ...` p1_argx p1_argy p1_argz
```

The command

```
p2 p2_arg1 p2_arg2 p2_arg3 …
```

will be executed as an independent command and the standard output from it will replace the whole backquoted string in the original command line. Then the modified command line will be executed.

E.g., if the standard output of

```
p2 p2_arg1 p2_arg2 p2_arg3 …
```

is

```
aaa bbb ccc
```

then the modified command line becomes

```
p1 p1_arg1 p1_arg2 aaa bbb ccc p1_argx p1_argy p1_argz
```

Which will then be executed.

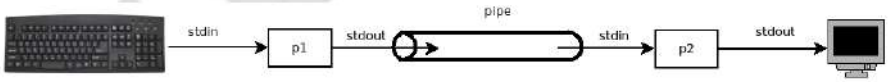
```
 x=10
y=20
z=`expr $x + $y`
+ expr 10 + 20
z=30
```

New Syntax

```
$ p1 p1_arg1 p1_arg2 $(p2 p2_arg1 p2_arg2 p2_arg3 ...) p1_argx
p1_argy p1_argz
```

Pipes

```
$ p1 | p2
```



Pipe

The processes p1 and p2 will be run concurrently as child processes of the shell, with the standard output of p1 redirected to a pipe and p2 taking its standard input from that pipe.

```
$ p1 | p2 | p3 | p4 | p5
```

p1 will take its standard input from the terminal (keyboard). Standard output of p1 will be supplied as standard input to p2. Standard output of p2 will be supplied as standard input to p3. Standard output of p3 will be supplied as standard input to p4. Standard output of p4 will be supplied as standard input to

p5. Standard output of p5 will appear on the terminal (monitor). In a chain of commands, each process carries out one specific stage of text processing.

Arguments for Different Redirection Operators

command1	>	file1
command1	>>	file1
Command1	2>	file1
command1	2>>	file1
command1	<	file1
command1	|	command2
command1	<<	WORD

Filters

A *filter* is a command that takes its input from standard input, processes it and produces the output on standard output. The benefit of a filter is that its input and output can be redirected independently and flexibly. Also, several filters can be chained in a pipeline.

```
cat a4 | grep '^[0-9]*,[4-9][0-9],'

cat in01.csv | grep PASS

cat in01.csv | grep PASS | cut -d, -f1,3

cat in01.csv | grep PASS | wc -l
```

```
cat in01.csv | grep PASS | cut -d, -f1,3 | sort -t, -k1,1n

SELECT no, name             # cut -d, -f1,3
    FROM students           # cat in01.csv
    WHERE result = 'PASS'   # grep PASS
    ORDER BY no             # sort -t, -k1,1n
```

```
cat orderdata.csv | sort -t, -k1,1n -k3,3nr

SELECT *
```

```
    FROM orderdata                              # cat orderdata
    ORDER BY orderno ASC, productno DESC        # sort -t, -k1,1n
-k3,3nr
```

Regular Expressions

Basic Regular Expressions

Pattern	Matches with
.	Any single character
*	Previous element 0 or more times
^	An imaginary anchor at the beginning of line (only when ^ is at the beginning of the pattern)
$	An imaginary anchor at the end of line (only when $ is at the end of the pattern)

\ is the *escape character* - it removes the special meaning of the next character; also adds special meaning to some characters in some circumstances as described below.

Character Classes

Pattern	Matches with
[abcd]	Any one character from a, b, c or d
[a-z]	Any one character between a and z, inclusive
[a-z0-9]	Any one character from between a and z, inclusive; or from between 0 and 9, inclusive
[a-zA-Z0-9]	Any one character from between a and z, inclusive; or from between A and Z, inclusive; or from between 0 and 9, inclusive
[a-z0-9ABCD]	Any one character from between a and z, inclusive or from between 0 and 9, inclusive or A or B or C or D
[^a-z0-9]	Any one character other than those between a and z, inclusive and those between 0 and 9, inclusive

Pattern	Matches with
[a-z^0-9]	Any one character from between a and z, inclusive or ^ or from between between 0 and 9, inclusive

Pattern	Matches with
\t	The TAB character
\n	The newline character
\\	The (backslash) character
\w	equivalent to [a-zA-Z0-9]
\W	equivalent to [^a-zA-Z0-9]

Extended Regular Expressions

The meta characters used in extended regular expressions are as follows. These meta characters can be used as given (individually) when extended regular expressions are enabled (e.g. grep -E). The grep command also supports them in basic regular expressions in their escaped form (e.g. \+ instead of +, \? instread of ?, \{ instread of {, etc.).

Pattern	Matches with
+	Previous element 1 or more times
?	Previous element 0 or 1 time

Some Common Filters / Commands

The grep Family of Commands

`grep` stands for *g*lobal *r*egular *e*xpression *p*rint. It is a pattern matching filter. It processes the input line-by-line and outputs only those lines where a part of the line or the whole line matches with the specified regular expression pattern. The `grep` command is used for horizontal (row / line) filtering. It works in a way somewhat similar to the `WHERE` clause of the SQL `SELECT` statement.

```
cat in01.csv
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | grep PASS
```

```
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | grep '^[0-9]*,[4-9][0-9],'
```

```
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
6,88,rr,PASS
13,55,12,PASS
```

```
cat in01.csv | grep -e '^[0-9]*,[4-9][0-9],' -e '^[0-9]*,100,'
```

```
2,45,bb,PASS
```

```
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

- **grep:** regular grep, did not support multiple patterns and extended regular expressions
- **egrep:** extended grep, supported multiple patterns and extended regular expressions
- **fgrep:** fast grep, did not support patterns (can search only for fixed strings), but supported multiple search strings

Note: under GNU/Linux, the functionalities of all three commands have been combined into a single command - `grep`
`grep -E` is equivalent to `egrep`
`grep -F` is equivalent to `fgrep`

The grep family of commands

Feature	grep	egrep (extended)	fgrep (fast)
Regular Expressions	Yes	Yes	No
Extended Regular Expressions	No	Yes	No
Multiple Patterns	No	Yes	Yes
Speed	Medium	Medium	Fast

The cut command

The `cut` command can be used to separate fields (columns) from rows / records / lines. It is used for vertical filtering. It works in a way similar to the column list in the SQL `SELECT` statement.

```
cat in01.csv
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | cut -d, -f1,3
```

```
1,aa
2,bb
3,cc
9,cc
10,cc
4,dd
11,dd
12,dd
5,ee
7,ff
8,ff
6,rr
13,12
999,er
```

```
cat in01.csv | cut -d, -f1-3
```

```
1,12,aa
2,45,bb
3,56,cc
9,44,cc
10,55,cc
4,67,dd
11,66,dd
12,55,dd
5,8,ee
7,12,ff
8,34,ff
```

```
6,88,rr
13,55,12
999,100,er
```

```
cat marks
```

```
01 55 59 61 60
02 45 37 51 23
03 67 63 71 79
04 63 65 60 58
05 08 21 43 32
06 32 45 54 55
07 45 55 50 57
```

```
cat marks | cut -d' ' -f2-4
```

```
55 59 61
45 37 51
67 63 71
63 65 60
08 21 43
32 45 54
45 55 50
```

```
cat marks | cut -d' ' -f1-3,5
```

```
01 55 59 60
02 45 37 23
03 67 63 79
04 63 65 58
05 08 21 32
06 32 45 55
07 45 55 57
```

```
cat /etc/passwd
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

```
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
saned:x:113:119::/var/lib/saned:/usr/sbin/nologin
pulse:x:114:120:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin
/nologin
jignesh:x:1000:1000:Jignesh,,,:/home/jignesh:/bin/bash
yash:x:1001:1001:Yash,,,:/home/yash:/bin/bash
```

```
cat /etc/passwd | cut -d: -f1,3,6,7
```

```
root:0:/root:/bin/bash
daemon:1:/usr/sbin:/usr/sbin/nologin
bin:2:/bin:/usr/sbin/nologin
sys:3:/dev:/usr/sbin/nologin
man:6:/var/cache/man:/usr/sbin/nologin
lp:7:/var/spool/lpd:/usr/sbin/nologin
mail:8:/var/mail:/usr/sbin/nologin
uucp:10:/var/spool/uucp:/usr/sbin/nologin
www-data:33:/var/www:/usr/sbin/nologin
backup:34:/var/backups:/usr/sbin/nologin
saned:113:/var/lib/saned:/usr/sbin/nologin
pulse:114:/var/run/pulse:/usr/sbin/nologin
jignesh:1000:/home/jignesh:/bin/bash
yash:1001:/home/yash:/bin/bash
```

```
cat /etc/group | grep jignesh
```

```
adm:x:4:syslog,jignesh
cdrom:x:24:jignesh
sudo:x:27:jignesh
dip:x:30:jignesh
plugdev:x:46:jignesh
lpadmin:x:116:jignesh
jignesh:x:1000:
sambashare:x:126:jignesh
vboxusers:x:127:jignesh
kvm:x:129:jignesh
libvirt:x:131:jignesh
docker:x:136:jignesh
```

```
cat /etc/group | grep jignesh | cut -d: -f1,4
```

```
adm:syslog,jignesh
cdrom:jignesh
sudo:jignesh
dip:jignesh
plugdev:jignesh
lpadmin:jignesh
jignesh:
sambashare:jignesh
vboxusers:jignesh
kvm:jignesh
libvirt:jignesh
docker:jignesh
```

```
cat in01.fw
```

```
00134 aaFAIL
  245 bbPASS
03 56 ccPASS
09 44 ccPASS
10 55 ccPASS
04 67 ddPASS
11 66 ddPASS
12 55 ddPASS
05 78 eeFAIL
07 12 ffFAIL
08 34 ffFAIL
06 88 rrPASS
13 55 12PASS
999100erPASS
```

```
cat in01.fw | cut -c1-3
```

```
001
  2
03
09
10
04
11
12
05
07
08
06
13
```

```
999
```

```
cat in01.fw | cut -c4-6
```

```
34
45
56
44
55
67
66
55
78
12
34
88
55
100
```

```
cat in01.fw | cut -c7-8
```

```
aa
bb
cc
cc
cc
dd
dd
dd
ee
ff
ff
rr
12
er
```

```
cat in01.fw | cut -c9-
```

```
FAIL
PASS
PASS
PASS
PASS
```

```
PASS
PASS
PASS
FAIL
FAIL
FAIL
PASS
PASS
PASS
```

```
cat in01.fw | cut -c1-3,9-
```

```
001FAIL
  2PASS
03 PASS
09 PASS
10 PASS
04 PASS
11 PASS
12 PASS
05 FAIL
07 FAIL
08 FAIL
06 PASS
13 PASS
999PASS
```

The `wc` (word count) command

The `wc` (word count) command can be used to count the number of lines, number of words and number of characters in the input. By default, it outputs all three. If we use the option `-l`, it outputs the number of lines. If we use the option `-w`, it outputs the number of words. If we use the option `-c`, it outputs the number of characters.

```
cat inlines
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
```

```
cat inlines | wc
```

```
      4      24     106
```

```
cat inlines | wc -l
```

```
4
```

```
cat inlines | wc -w
```

```
24
```

```
cat inlines | wc -c
```

```
106
```

```
cat in01.csv | wc
```

```
     14      14     188
```

```
cat in01.csv | wc -l
```

```
14
```

```
cat in01.csv | grep PASS | wc -l
```

```
10
```

```
cat in01.csv | grep FAIL | wc -l
```

```
4
```

The sort command

The `sort` command can be used to sort (arrange) the input records / rows / lines. It is similar to the `ORDER BY` clause of the SQL `SELECT` statement. Sorting can be done on one or more keys. The delimiter / separator character is specified using the `-t` option. The keys are specified using the syntax `-k1,1` where `1` is the position of the field on which sorting is to be done. The default sorting method is to use the dictionary order. For numeric sorting, the `-n` option can be used. The default sort order is ascending. For descending order, the `-r` option may be used.

```
cat in01.csv
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sort -t, -k1,1
```

```
1,12,aa,FAIL
10,55,cc,PASS
11,66,dd,PASS
12,55,dd,PASS
13,55,12,PASS
2,45,bb,PASS
3,56,cc,PASS
4,67,dd,PASS
5,8,ee,FAIL
6,88,rr,PASS
```

```
7,12,ff,FAIL
8,34,ff,FAIL
9,44,cc,PASS
999,100,er,PASS
```

```
cat in01.csv | sort -t, -k1,1n
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
4,67,dd,PASS
5,8,ee,FAIL
6,88,rr,PASS
7,12,ff,FAIL
8,34,ff,FAIL
9,44,cc,PASS
10,55,cc,PASS
11,66,dd,PASS
12,55,dd,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sort -t, -k1,1nr
```

```
999,100,er,PASS
13,55,12,PASS
12,55,dd,PASS
11,66,dd,PASS
10,55,cc,PASS
9,44,cc,PASS
8,34,ff,FAIL
7,12,ff,FAIL
6,88,rr,PASS
5,8,ee,FAIL
4,67,dd,PASS
3,56,cc,PASS
2,45,bb,PASS
1,12,aa,FAIL
```

```
cat in01.csv | sort -t, -k2,2n
```

```
5,8,ee,FAIL
1,12,aa,FAIL
```

```
7,12,ff,FAIL
8,34,ff,FAIL
9,44,cc,PASS
2,45,bb,PASS
10,55,cc,PASS
12,55,dd,PASS
13,55,12,PASS
3,56,cc,PASS
11,66,dd,PASS
4,67,dd,PASS
6,88,rr,PASS
999,100,er,PASS
```

```
cat in01.csv | sort -t, -k2,2n -k3,3
```

```
5,8,ee,FAIL
1,12,aa,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
9,44,cc,PASS
2,45,bb,PASS
13,55,12,PASS
10,55,cc,PASS
12,55,dd,PASS
3,56,cc,PASS
11,66,dd,PASS
4,67,dd,PASS
6,88,rr,PASS
999,100,er,PASS
```

```
cat in01.csv | sort -t, -k2,2n -k3,3r
```

```
5,8,ee,FAIL
7,12,ff,FAIL
1,12,aa,FAIL
8,34,ff,FAIL
9,44,cc,PASS
2,45,bb,PASS
12,55,dd,PASS
10,55,cc,PASS
13,55,12,PASS
3,56,cc,PASS
11,66,dd,PASS
4,67,dd,PASS
6,88,rr,PASS
999,100,er,PASS
```

The tr (translate) command

The `tr` (translate) command operates on individual characters in the input. It can be used to translate (convert) characters, to delete characters as well as to squeezee characters from the input. To squeeze a character means multiple consecutive occurrences of that character are converted into a single occurrence.

```
cat inwords
```

```
at   begin

presentation file
in  cat     hat

document file
chat   industry

spreadsheet file
```

```
cat inwords | tr 'abcdefghijklmnopqrstuvwxyz'
        'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
AT   BEGIN

PRESENTATION FILE
IN  CAT     HAT
```

```
DOCUMENT FILE
CHAT   INDUSTRY

SPREADSHEET FILE
```

```
cat inwords | tr 'a-z' 'A-Z'
```

```
AT   BEGIN

PRESENTATION FILE
IN  CAT     HAT

DOCUMENT FILE
CHAT   INDUSTRY

SPREADSHEET FILE
```

```
cat inwords | tr 'a-z' 'b-za'
```

```
bu   cfhjo

qsftfoubujpo gjmf
jo  dbu     ibu

epdvnfou gjmf
dibu   joevtusz

tqsfbetiffu gjmf
```

```
cat inwords | tr -d 'a'
```

```
t   begin

presenttion file
in  ct     ht

document file
cht   industry

spredsheet file
```

```
cat inwords | tr -d 'aeiou'
```

```
t   bgn

prsnttn fl
n  ct     ht

dcmnt fl
cht   ndstry

sprdsht fl
```

```
cat inwords | tr -d 'a-j'
```

```
t   n

prsntton l
n  t     t

oumnt l
t   nustry

sprst l
```

```
 ls -l
```

```
total 108
-rwxrwxr-x 1 jignesh jignesh   20 Feb  1 23:10 in
-rwxrwxr-x 1 jignesh jignesh  118 Feb  1 23:10 in00.csv
-rwxrwxr-x 1 jignesh jignesh    9 Feb  1 23:10 in01
-rwxrwxr-x 1 jignesh jignesh  189 Feb  1 23:10 in01b.csv
-rwxrwxr-x 1 jignesh jignesh  188 Feb  1 23:10 in01.csv
-rwxrwxr-x 1 jignesh jignesh  182 Feb  1 23:10 in01.fw
-rwxrwxr-x 1 jignesh jignesh 1266 Feb  1 23:10 in01.json
-rw-r--r-- 1 jignesh jignesh  188 Feb  1 23:10 in01.space.txt
-rwxrwxr-x 1 jignesh jignesh 1421 Feb  1 23:10 in01.xml
-rwxrwxr-x 1 jignesh jignesh  785 Feb  1 23:10 in02
-rwxrwxr-x 1 jignesh jignesh   66 Feb  1 23:10 in03
-rwxrwxr-x 1 jignesh jignesh   36 Feb  1 23:10 in03a
-rwxrwxr-x 1 jignesh jignesh   10 Feb  1 23:10 in04
-rwxrwxr-x 1 jignesh jignesh  208 Feb  1 23:10 in04.csv
-rwxrwxr-x 1 jignesh jignesh  195 Feb  1 23:10 in05
-rwxrwxr-x 1 jignesh jignesh   28 Feb  1 23:10 in-amounts
-rwxrwxr-x 1 jignesh jignesh   72 Feb  1 23:10 in-case-sensitive
```

```
-rwxrwxr-x 1 jignesh jignesh  180 Feb  1 23:10 inhtml
-rwxrwxr-x 1 jignesh jignesh  106 Feb  1 23:10 inlines
-rwxrwxr-x 1 jignesh jignesh   51 Feb  1 23:10 inmanyfields
-rwxrwxr-x 1 jignesh jignesh   45 Feb  1 23:10 inmixedcase
-rwxrwxr-x 1 jignesh jignesh  193 Feb  1 23:10 innum01
-rwxrwxr-x 1 jignesh jignesh   20 Feb  1 23:10 input
-rwxrwxr-x 1 jignesh jignesh  783 Feb  1 23:10 in-random-numbers
-rw-rw-r-- 1 jignesh jignesh  220 Feb  1 23:10 integer-
comparison.sh
-rwxrwxr-x 1 jignesh jignesh  205 Feb  1 23:10 invalid
-rwxrwxr-x 1 jignesh jignesh   95 Feb  1 23:10 inwords
```

```
ls -l | cut -d' ' -f5
```

```
1266

1421
```

```
ls -l | tr -s ' ' | cut -d' ' -f5
```

```
20
118
9
```

```
189
188
182
1266
188
1421
785
66
36
10
208
195
28
72
180
106
51
45
193
20
783
220
205
95
```

The sed (stream editor) filter

The `sed` (stream editor) command is used to edit (modify) its input or a file programmatically from a script. It does not modify the input file. Rather, it processes the input line-by-line and produces the output on the standard output. The standard output can be redirected to another file. That file may be moved on to the original file, if desired.

`sed` processes the input line-by-line according to one or more `sed` commands or `sed` script given to it. By default, `sed` output each line from the input after processing. This default processing can be disabled using the option `-n`. The common `sed` commands are `p` (print), `s` (substitute i.e. replace), `d` (delete), `i` (insert) and `a` (append). By default, these commands apply to all input lines. However, if a line number is specified before the command, then the commands apply only to that line. If a range of lines (starting line number, ending line number) is specified before the command, then the commands apply only to that range of lines. Instead of line numbers, regular expression patterns can also be specified using the syntax `/pattern/`.

```
cat inlines
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
```

```
cat inlines | sed 'p'
```

```
It was the best of times
It was the best of times
It was the worst of times
It was the worst of times
It was the age of wisdom
It was the age of wisdom
It was the age of foolishness
It was the age of foolishness
```

```
cat inlines | sed -n 'p'
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
```

```
cat inlines | sed -n '1p'
```

```
It was the best of times
```

```
cat inlines | sed -n '1,3p'
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
```

```
cat inlines | sed -n '/times/p'
```

```
It was the best of times
It was the worst of times
```

```
cat inlines | sed -n '/times/,/wisdom/p'
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
```

```
cat in01a.csv
```

```
1,12,aa
2,45,bb
3,56,cc
9,44,cc
10,55,cc
4,67,dd
11,66,dd
12,55,dd
5,8,ee
7,12,ff
8,34,ff
6,88,rr
13,55,12
999,100,er
```

```
cat in01a.csv | sed '/^[0-9]*,[4-9][0-9],/s/$/,PASS/'
```

```
1,12,aa
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee
7,12,ff
8,34,ff
6,88,rr,PASS
13,55,12,PASS
999,100,er
```

```
cat in01a.csv | sed -e '/^[0-9]*,[4-9][0-9],/s/$/,PASS/' -e
        '/^[0-9]*,100,/s/$/,PASS/'
```

```
1,12,aa
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee
7,12,ff
8,34,ff
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01a.csv | sed -e '/^[0-9]*,[4-9][0-9],/s/$/,PASS/' -e
        '/^[0-9]*,100,/s/$/,PASS/' -e '/^[0-9]*,[0-3][0-9],
        /s/$/,FAIL/'
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01a.csv | sed -e '/^[0-9]*,[4-9][0-9],/s/$/,PASS/' -e
        '/^[0-9]*,100,/s/$/,PASS/' -e '/^[0-9]*,[0-3][0-9],
        /s/$/,FAIL/' -e '/^[0-9]*,[0-9],/s/$/,FAIL/'
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
```

```
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat sedf-result
```

```
/^[0-9]*,[4-9][0-9],/s/$/,PASS/
/^[0-9]*,100,/s/$/,PASS/
/^[0-9]*,[0-3][0-9],/s/$/,FAIL/
/^[0-9]*,[0-9],/s/$/,FAIL/
```

```
cat in01a.csv | sed -f sedf-result
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
```

```
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sed '1d'
```

```
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sed '1,7d'
```

```
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sed '/FAIL/d'
```

```
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
11,66,dd,PASS
```

```
12,55,dd,PASS
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sed '/cc/,/rr/d'
```

```
1,12,aa,FAIL
2,45,bb,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat in01.csv | sed '/PASS/i\
Congratulations!\
You did it!'
```

```
1,12,aa,FAIL
Congratulations!
You did it!
2,45,bb,PASS
Congratulations!
You did it!
3,56,cc,PASS
Congratulations!
You did it!
9,44,cc,PASS
Congratulations!
You did it!
10,55,cc,PASS
Congratulations!
You did it!
4,67,dd,PASS
Congratulations!
You did it!
11,66,dd,PASS
Congratulations!
You did it!
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
Congratulations!
You did it!
6,88,rr,PASS
Congratulations!
You did it!
```

```
13,55,12,PASS
Congratulations!
You did it!
999,100,er,PASS
```

```
cat in01.csv | sed '/PASS/a\
Congratulations!\
You did it!'
```

```
1,12,aa,FAIL
2,45,bb,PASS
Congratulations!
You did it!
3,56,cc,PASS
Congratulations!
You did it!
9,44,cc,PASS
Congratulations!
You did it!
10,55,cc,PASS
Congratulations!
You did it!
4,67,dd,PASS
Congratulations!
You did it!
11,66,dd,PASS
Congratulations!
You did it!
12,55,dd,PASS
Congratulations!
You did it!
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
Congratulations!
You did it!
13,55,12,PASS
Congratulations!
You did it!
999,100,er,PASS
Congratulations!
You did it!
```

```
cat in01.csv | sed -e '/PASS/a\
Congratulations!\
You did it!' -e '/FAIL/a\
```

```
Sorry.\
Better luck next time.'
```

```
1,12,aa,FAIL
Sorry.
Better luck next time.
2,45,bb,PASS
Congratulations!
You did it!
3,56,cc,PASS
Congratulations!
You did it!
9,44,cc,PASS
Congratulations!
You did it!
10,55,cc,PASS
Congratulations!
You did it!
4,67,dd,PASS
Congratulations!
You did it!
11,66,dd,PASS
Congratulations!
You did it!
12,55,dd,PASS
Congratulations!
You did it!
5,8,ee,FAIL
Sorry.
Better luck next time.
7,12,ff,FAIL
Sorry.
Better luck next time.
8,34,ff,FAIL
Sorry.
Better luck next time.
6,88,rr,PASS
Congratulations!
You did it!
13,55,12,PASS
Congratulations!
You did it!
999,100,er,PASS
Congratulations!
You did it!
```

```
cat sedf-append
```

```
/PASS/a\
Congratulations!\
You did it!
/FAIL/a\
Sorry.\
Better luck next time.
```

```
cat in01.csv | sed -f sedf-append
```

```
1,12,aa,FAIL
Sorry.
Better luck next time.
2,45,bb,PASS
Congratulations!
You did it!
3,56,cc,PASS
Congratulations!
You did it!
9,44,cc,PASS
Congratulations!
You did it!
10,55,cc,PASS
Congratulations!
You did it!
4,67,dd,PASS
Congratulations!
You did it!
11,66,dd,PASS
Congratulations!
You did it!
12,55,dd,PASS
Congratulations!
You did it!
5,8,ee,FAIL
Sorry.
Better luck next time.
7,12,ff,FAIL
Sorry.
Better luck next time.
8,34,ff,FAIL
Sorry.
Better luck next time.
6,88,rr,PASS
Congratulations!
You did it!
13,55,12,PASS
Congratulations!
You did it!
```

```
999,100,er,PASS
Congratulations!
You did it!
```

The awk filter

The `awk` command is named after its authors Aho, Kernighan, and Weinberger. It is like the Swiss army knife of text processing. It combines features of several other filters into a single command. It has its own built-in programming language with variables, control structures, functions, etc. The programming language is case-sensitive. Its syntax is very similar to the `C` programming language.

`awk` processes its input line-by-line according to the `awk` script given to it. The `awk` script consists of a number of `pattern, action` pairs. The action is written in `{ }` (curly braces). The action is executed on the lines that match the pattern.

```
pattern1    { action1 }
pattern2    { action2 }
pattern3    { action3 }
```

Both pattern and action are optional and any one can be omitted, but not both for obvious reasons. If the pattern is skipped, the action is executed for each and every line of the input. If the action is omitted, the default action is to print the whole line.

There are two special patterns. The action associated with the special pattern `BEGIN` is executed at the beginning, before the processing of the input lines begins. The action associated with the special pattern `END` is executed at the end, after all the input lines have been processed.

As separating fields from the input is the most common text processing task, `awk` automatically separates fields from the current input line according to the delimiter specified in the built-in variable `FS` and stores the fields in variables `$1`, `$2`, `$3`, etc. There is no limit on the number of fields in a line. Different lines can have different number of fields too. The variable `$0` represents the whole line.

`awk` has a built-in `print` statement to print one or more items. The items are output, separated by the output field separator character stored in the `OFS` built-in variable.

- `awk` built-in variables
  - **FS** Field separator
  - **NF** Number of fields in the current line
  - **NR** Number of Record (number of lines processed so far)
  - **OFS** Output field separator (inserted between fields by print)

```
cat marks
```

```
01 55 59 61 60
02 45 37 51 23
03 67 63 71 79
04 63 65 60 58
05 08 21 43 32
06 32 45 54 55
07 45 55 50 57
```

```
cat marks | awk '{print "$1=" $1 " $2=" $2 " $3=" $3 " $4=" $4 "
        $5=" $5}'
```

```
$1=01 $2=55 $3=59 $4=61 $5=60
$1=02 $2=45 $3=37 $4=51 $5=23
$1=03 $2=67 $3=63 $4=71 $5=79
$1=04 $2=63 $3=65 $4=60 $5=58
$1=05 $2=08 $3=21 $4=43 $5=32
$1=06 $2=32 $3=45 $4=54 $5=55
$1=07 $2=45 $3=55 $4=50 $5=57
```

```
cat inmanyfields
```

```
111 222 333 444 55 66 77 88 99 1010 1111 1212 1313
```

```
cat inmanyfields | awk '{print $1, $2, $3, $4, $5, $6, $7, $8,
        $9, $10, $11, $12, $13}'
```

```
111 222 333 444 55 66 77 88 99 1010 1111 1212 1313
```

```
cat inlines
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
```

```
cat inlines | awk 'BEGIN {print "Beginning...";}
{ print $0;}
END {print "end...";}'
```

```
Beginning...
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
end...
```

```
cat awkf03
```

```
# input file: any
BEGIN {print "Beginning...";}
{ print $0;}
END {print "end...";}
```

```
cat inlines | awk -f awkf03
```

```
Beginning...
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
end...
```

```
cat in01.csv
```

```
1,12,aa,FAIL
2,45,bb,PASS
3,56,cc,PASS
9,44,cc,PASS
10,55,cc,PASS
4,67,dd,PASS
```

```
11,66,dd,PASS
12,55,dd,PASS
5,8,ee,FAIL
7,12,ff,FAIL
8,34,ff,FAIL
6,88,rr,PASS
13,55,12,PASS
999,100,er,PASS
```

```
cat awkf04
```

```
# input file: any
BEGIN {count=0;}
{ count++;}
END {print count;}
```

```
cat in01.csv | awk -f awkf04
```

```
14
```

```
cat awkf05
```

```
# input file: any
END {print NR;}
```

```
cat in01.csv | awk -f awkf05
```

```
14
```

```
cat in01.space.txt
```

```
1 12 aa FAIL
2 45 bb PASS
3 56 cc PASS
9 44 cc PASS
10 55 cc PASS
4 67 dd PASS
11 66 dd PASS
```

```
12 55 dd PASS
5 8 ee FAIL
7 12 ff FAIL
8 34 ff FAIL
6 88 rr PASS
13 55 12 PASS
999 100 er PASS
```

```
cat awkf06
```

```
#input file: in01.space.txt
{print "no=", $1, "marks=", $2, "name=", $3, "result=", $4}
```

```
cat in01.space.txt | awk -f awkf06
```

```
no= 1 marks= 12 name= aa result= FAIL
no= 2 marks= 45 name= bb result= PASS
no= 3 marks= 56 name= cc result= PASS
no= 9 marks= 44 name= cc result= PASS
no= 10 marks= 55 name= cc result= PASS
no= 4 marks= 67 name= dd result= PASS
no= 11 marks= 66 name= dd result= PASS
no= 12 marks= 55 name= dd result= PASS
no= 5 marks= 8 name= ee result= FAIL
no= 7 marks= 12 name= ff result= FAIL
no= 8 marks= 34 name= ff result= FAIL
no= 6 marks= 88 name= rr result= PASS
no= 13 marks= 55 name= 12 result= PASS
no= 999 marks= 100 name= er result= PASS
```

```
cat awkf07
```

```
#input file: in01.csv
BEGIN {FS=",";}
{print "no=", $1, "marks=", $2, "name=", $3, "result=", $4}
```

```
cat in01.csv | awk -f awkf07
```

```
no= 1 marks= 12 name= aa result= FAIL
no= 2 marks= 45 name= bb result= PASS
no= 3 marks= 56 name= cc result= PASS
```

```
no= 9 marks= 44 name= cc result= PASS
no= 10 marks= 55 name= cc result= PASS
no= 4 marks= 67 name= dd result= PASS
no= 11 marks= 66 name= dd result= PASS
no= 12 marks= 55 name= dd result= PASS
no= 5 marks= 8 name= ee result= FAIL
no= 7 marks= 12 name= ff result= FAIL
no= 8 marks= 34 name= ff result= FAIL
no= 6 marks= 88 name= rr result= PASS
no= 13 marks= 55 name= 12 result= PASS
no= 999 marks= 100 name= er result= PASS
```

```
cat awkf08
```

```
# input file: inwords, inlines
BEGIN {count=0;}
{count += NF;}
END {print count;}
```

```
cat inlines
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
```

```
cat inlines | awk -f awkf08
```

```
24
```

```
cat inwords
```

```
at   begin

presentation file
in  cat     hat

document file
chat   industry

spreadsheet file
```

```
cat inwords | awk -f awkf08
```

```
13
```

```
cat awkf09
```

```
#input file: in01.space.txt (Default OFS)
{print "no=", $1, "marks=", $2, "name=", $3, "result=", $4}
```

```
cat in01.space.txt | awk -f awkf09
```

```
no= 1 marks= 12 name= aa result= FAIL
no= 2 marks= 45 name= bb result= PASS
no= 3 marks= 56 name= cc result= PASS
no= 9 marks= 44 name= cc result= PASS
no= 10 marks= 55 name= cc result= PASS
no= 4 marks= 67 name= dd result= PASS
no= 11 marks= 66 name= dd result= PASS
no= 12 marks= 55 name= dd result= PASS
no= 5 marks= 8 name= ee result= FAIL
no= 7 marks= 12 name= ff result= FAIL
no= 8 marks= 34 name= ff result= FAIL
no= 6 marks= 88 name= rr result= PASS
no= 13 marks= 55 name= 12 result= PASS
no= 999 marks= 100 name= er result= PASS
```

```
cat awkf10
```

```
# input file: in01.space.txt
BEGIN {OFS=":"}
```

```
{print "no=", $1, "marks=", $2, "name=", $3, "result=", $4}
```

```
cat in01.space.txt | awk -f awkf10
```

```
no=:1:marks=:12:name=:aa:result=:FAIL
no=:2:marks=:45:name=:bb:result=:PASS
no=:3:marks=:56:name=:cc:result=:PASS
no=:9:marks=:44:name=:cc:result=:PASS
no=:10:marks=:55:name=:cc:result=:PASS
no=:4:marks=:67:name=:dd:result=:PASS
no=:11:marks=:66:name=:dd:result=:PASS
no=:12:marks=:55:name=:dd:result=:PASS
no=:5:marks=:8:name=:ee:result=:FAIL
no=:7:marks=:12:name=:ff:result=:FAIL
no=:8:marks=:34:name=:ff:result=:FAIL
no=:6:marks=:88:name=:rr:result=:PASS
no=:13:marks=:55:name=:12:result=:PASS
no=:999:marks=:100:name=:er:result=:PASS
```

```
cat awkf11
```

```
# input file: in01.csv
BEGIN {FS=",";OFS=":"}
{print "no=", $1, "marks=", $2, "name=", $3, "result=", $4}
```

```
cat in01.csv | awk -f awkf11
```

```
no=:1:marks=:12:name=:aa:result=:FAIL
no=:2:marks=:45:name=:bb:result=:PASS
no=:3:marks=:56:name=:cc:result=:PASS
no=:9:marks=:44:name=:cc:result=:PASS
no=:10:marks=:55:name=:cc:result=:PASS
no=:4:marks=:67:name=:dd:result=:PASS
no=:11:marks=:66:name=:dd:result=:PASS
no=:12:marks=:55:name=:dd:result=:PASS
no=:5:marks=:8:name=:ee:result=:FAIL
no=:7:marks=:12:name=:ff:result=:FAIL
no=:8:marks=:34:name=:ff:result=:FAIL
no=:6:marks=:88:name=:rr:result=:PASS
no=:13:marks=:55:name=:12:result=:PASS
no=:999:marks=:100:name=:er:result=:PASS
```

```
cat marks
```

```
01 55 59 61 60
02 45 37 51 23
03 67 63 71 79
04 63 65 60 58
05 08 21 43 32
06 32 45 54 55
07 45 55 50 57
```

```
cat awkf12
```

```
# input file: marks
{ total=$2+$3+$4+$5;
printf "%s total=%d\n", $0, total;}
```

```
cat marks | awk -f awkf12
```

```
01 55 59 61 60 total=235
02 45 37 51 23 total=156
03 67 63 71 79 total=280
04 63 65 60 58 total=246
05 08 21 43 32 total=104
06 32 45 54 55 total=186
07 45 55 50 57 total=207
```

```
cat awkf13
```

```
# input file: in01.csv
BEGIN {
        FS=",";
    }
    {
        result[$1]=$4;
        printf "result[%d]=%s\n", $1, $4;
    }
END {
        printf "|-----|------|\n" ;
        for (studno in result) {
            printf "| %03d | %4s |\n", studno, result[studno];
        }
```

```
        printf "|-----|------|\n" ;
    }
```

```
cat in01.csv | awk -f awkf13
```

```
result[1]=FAIL
result[2]=PASS
result[3]=PASS
result[9]=PASS
result[10]=PASS
result[4]=PASS
result[11]=PASS
result[12]=PASS
result[5]=FAIL
result[7]=FAIL
result[8]=FAIL
result[6]=PASS
result[13]=PASS
result[999]=PASS
|-----|------|
| 001 | FAIL |
| 002 | PASS |
| 003 | PASS |
| 004 | PASS |
| 005 | FAIL |
| 006 | PASS |
| 007 | FAIL |
| 008 | FAIL |
| 009 | PASS |
| 010 | PASS |
| 011 | PASS |
| 012 | PASS |
| 013 | PASS |
| 999 | PASS |
|-----|------|
```

```
cat awkf14
```

```
# input file: none
BEGIN {
a[5]="aaaa";
a[10]="bbbb";
a["cccc"]=15;
a["dddd"]="eeee";
a["ffff"]=20;
for (key in a) {
```

```
printf "a[%s]=%s\n", key, a[key];
}
printf "a[12]=XXX%sYYY\n", a[12];
a["cccc"]="gggg";
printf "a[cccc]=%s\n", a["cccc"];
a["ffff"]++;
printf "a[ffff]=%s\n", a["ffff"];
a["cccc"]++;
printf "a[cccc]=%s\n", a["cccc"];
delete a["cccc"];
printf "a[cccc]=%s\n", a["cccc"];
}
```

```
awk -f awkf14
```

```
a[cccc]=15
a[ffff]=20
a[dddd]=eeee
a[5]=aaaa
a[10]=bbbb
a[12]=XXXYYY
a[cccc]=gggg
a[ffff]=21
a[cccc]=1
a[cccc]=
```

```
cat awkf15
```

```
# input file: inlines
{ for (i=1; i<=NF; i++) freq[$i]++; }
END { for (word in freq)
print word, "->", freq[word];
}
```

```
cat inlines
```

```
It was the best of times
It was the worst of times
It was the age of wisdom
It was the age of foolishness
```

```
cat inlines | awk -f awkf15
```

```
age -> 2
was -> 4
best -> 1
worst -> 1
wisdom -> 1
of -> 4
the -> 4
It -> 4
foolishness -> 1
times -> 2
```

Commands List

- **File System Handling**
  - **pwd** Displays the present working directory
  - **ls** Displays list of files
    - **-d** When an argument is a directory, display the directory (default is to display the contents of the directory)
    - **-h** Display human-readable size (20M, 1.5G, etc.)
    - **-l** Display a long list
  - **cat** Output standard input / a file / concatenate multiple files and output
  - **cd** Change directory
  - **mkdir** Make one or more directories
  - **rmdir** Remove one or more directories (must be empty)
  - **rm** Remove file(s) (and directories)
    - **-R** Remove file(s) and directories recursively
    - **-i** Remove interactively (prompt before deleting each file/directory)
  - **cp** Copy file(s)
    - **-R** Copy file(s) and directories recursively
    - **-i** Copy interactively (prompt before deleting each file/directory)
  - **mv** Move/rename files and directories
    - **-i** Move/rename files and directories interactively (prompt before deleting each file/directory)
- **Editing**
  - **nano**
- **Process Management**
  - **ps** Process Status; display a list of running processes (by default, only processes in the current session)
    - **-a** (all) processes belonging to the current user sessions except

the session leaders (like the shell process that is the root of the process hierarchy for the session)
    - **-e** (every) Displays all the interactive as well as non-interactive processes of all the users, including the system processes
    - **-f** (full) Show the full information including parent process id and the full command line
  - **kill** Send a signal to another process
    - **-9** Send the kill signal to another process to kill it
- **Basic Filters**
  - **cut** Extract one or more fields (columns) from input
    - **-c** Specify column positions (for use with fixed-width input)
    - **-d** Specify delimiter (for use with delimited input)
    - **-f** Specify fields to be extracted (for use with delimited input)
  - **grep (egrep, fgrep)**
    - **-E** Enable extended regular expression
    - **-e** Specify a pattern (used to specify multiple patterns)
    - **-F** Use fixed strings (work like fgrep)
    - **-f** Specify a file containing patterns, one per line
    - **-i** Ignore case when matching patterns
    - **-c** Output a count of matching lines
    - **-v** Invert match, output lines other than those having a match for the pattern(s)
    - **-w** word match, pattern must match with a whole word in the line
    - **-l** Output only a list of files that have one or more matches
    - **-n** Output line numbers along with matching lines
    - **--color=always|none|auto** Produce color output always/never /decide automatically based on the standard output destination
  - **wc** Word Count; count the number of lines, words and characters in standard input or a file
    - **-l** Output line count
    - **-w** Output line count
    - **-c** Output line count
  - **head** Display the first few lines of a file or standard input (10 by default)
    - **-n** number of lines from the beginning to be displayed
  - **tail** Display the last few lines of a file or standard input (10 by default)
    - **-n** number of lines from the end to be displayed
  - **sort** Sort the input
    - **-k** Specify key (starting column, ending column) on which to sort.

In case of delimited input, specify starting and ending field number, e.g. -k2,2. In case of fixed-width input, specify 1.starting character position and 1.ending character position, e.g. -k1.5,1.10 May be repeated for sorting on multiple keys
    - **-r** sort in reverse (descending) order
    - **-t** Specify the delimiter
    - **-f** Fold lowercase to uppercase (ignore case while sorting)
    - **-n** Sort numerically
    - **-h** Sort numerically, human readable sizes (K, M, G, etc.)
  - **uniq** Output unique or repeated lines (assumes the input is sorted)
    - **-c** Prefix lines by a count of how many times the line is repeated
    - **-d** Only output duplicated (repeated) lines
    - **-i** ignore case
    - **-u** Only output unique lines (default)
  - **tr** Translate - convert or delete individual characters
    - **-s** Squeeze - convert multiple consecutive occurrences of the given characters into a single occurrence
    - **-d** Delete the given characters from the input
- **Other Commands**
  - **clear** clear the screen
  - **more** displays a file or standard input one page at a time. Press ENTER for next line, SPACE for next page, b for previous page (not supported with pipes), q to quit, /patternENTER search, n/N to search again in forward/backward direction
  - **less** More powerful version of more, supports two-way scrolling with pipes as well
  - **echo** outputs its arguments followed by a newline
    - **-n** do not output a newline at the end
  - **read** Input a line and assign it to a variable
    - **-p** display the argument as a prompt if the input comes from a terminal
  - **test** test various conditions
    - **integer tests**
      - › ***no1*** **-gt** ***no2*** True if *no1* is greater than *no2*
      - › ***no1*** **-ge** ***no2*** True if *no1* is greater than or equal to *no2*
      - › ***no1*** **-lt** ***no2*** True if *no1* is less than *no2*
      - › ***no1*** **-le** ***no2*** True if *no1* is less than or equal to *no2*
      - › ***no1*** **-eq** ***no2*** True if *no1* is equal to *no2* (integer comparison)
      - › ***no1*** **-ne** ***no2*** True if *no1* is not equal to *no2* (integer

comparison)

- **man** show the manual page for the argument from the first section in which it exists

- Advanced Filters
  - **sed** Stream editor. Used to edit a file programmatically (from script)
    - **-n** Do not perform default output
    - **-f** Specify sed script filename
    - **commands**
      - **s** Substitute
      - **i** Insert line(s) before
      - **a** Append line(s) after
      - **d** delete line
      - **p** print (output) line
  - **awk** Powerful data manipulation program
    - **-f** Specify awk script filename
    - **Built-in Variables**
      - **FS** Field separator
      - **NF** Number of fields in the current line
      - **NR** Number of Record (a cumulative line number for all the lines from all the files processed so far)
      - **OFS** Output field separator (inserted between fields by print)