**Chapter 1. Shell Something Out**

# Introduction

$ represents regular users and # represents the administrative user root.

Shebang is a line on which #! is prefixed to the interpreter path. /bin/bash is the interpreter command path for Bash.

When a shell is started, it initially executes a set of commands to define various settings such as prompt text, colors, and much more. This set of commands are read from a shell script at ~/.bashrc (or ~/.bash\_profile for login shells) located in the home directory of the user. The Bash shell also maintains a history of commands run by the user. It is available in the ~/.bash\_history file.

~ denotes your home directory, which is usually /home/user

**$ cmd1 ; cmd2**

This is equivalent to:

**$ cmd1**

**$ cmd2**

# Printing in the terminal

As with PS: ‘ ‘ is literal, “ “ will use special characters, unless escaped with \

printf takes quoted text or arguments delimited by spaces. We can use formatted strings with printf. We can specify string width, left or right alignment, and so on.

%s, %c, %d, and %f are format substitution characters for which an argument can be placed after the quoted format string.

%-5s can be described as a string substitution with left alignment (- represents left alignment) with width equal to 5. If - was not specified, the string would have been aligned to the right. The width specifies the number of characters reserved for that variable. For Name, the width reserved is 10. Hence, any name will reside within the 10-character width reserved for it and the rest of the characters will be filled with space up to 10 characters in total.

For floating point numbers, we can pass additional parameters to round off the decimal places.

For marks, we have formatted the string as %-4.2f, where .2 specifies rounding off to two decimal places. Note that for every line of the format string a newline (\n) is issued.

While using flags for echo and printf, always make sure that the flags appear before any strings in the command, otherwise Bash will consider the flags as another string.

echo –e (use escape sequences) –n (avoid newline at end)

Colors are represented by color codes, some examples being, reset = 0, black = 30, red = 31, green = 32, yellow = 33, blue = 34, magenta = 35, cyan = 36, and white = 37.

For a colored background, reset = 0, black = 40, red = 41, green = 42, yellow = 43, blue = 44, magenta = 45, cyan = 46, and white=47, are the color codes that are commonly used.

# Playing with variables and environment variables

In Bash, the value for every variable is string, regardless of whether we assign variables with quotes or without quotes.

We can obtain the process ID of gedit with the pgrep command as follows:

**$ pgrep gedit**

**12501**

null character (\0)

Linux Shell is case sensitive <- booooo….ooooooo

And it’s space sensitive, goddamit Linux

Some of the well-known environment variables are HOME, PWD, USER, UID, SHELL

**${#var}** returns the length of a string

To identify the shell which is currently being used, we can use the SHELL variable, like so:

**echo $SHELL**

Or:

**echo $0**

# Function to prepend to environment variables

APPEND:

**export PATH=$PATH:/opt/myapp/bin**

PREPEND:

**export PATH=/opt/myapp/bin:$PATH**

# Math with the shell

The Bash shell environment can perform basic arithmetic operations using the commands let, (( )), and []. The two utilities expr and bc are also very helpful in performing advanced operations.

While using let, we use variable names without the $ prefix

* Increment operation:
* **$ let no1++**
* Decrement operation:
* **$ let no1--**
* Shorthands:
* **let no+=6**

**let no-=6**

* Alternate methods:

The [] operator can be used in the same way as the let command as follows:

**result=$[ no1 + no2 ]**

Using the $ prefix inside [] operators are legal, for example:

**result=$[ $no1 + 5 ]**

(( )) can also be used. $ prefixed with a variable name is used when (( )) operator is used, as follows:

**result=$(( no1 + 50 ))**

expr can also be used for basic operations:

**result=`expr 3 + 4`**

**result=$(expr $no1 + 5)**

All of the preceding methods do not support floating point numbers, and operate on integers only.

Additional parameters can be passed to bc with prefixes to the operation with semicolon as delimiters through stdin.

* **Decimal places scale with bc**: In the following example the scale=2 parameter sets the number of decimal places to 2. Hence, the output of bc will contain a number with two decimal places:
* **echo "scale=2;3/8" | bc**
* **0.37**
* **Base conversion with bc**: We can convert from one base number system to another one. Let us convert from decimal to binary, and binary to octal:
* **#!/bin/bash**
* **Desc: Number conversion**
* **no=100**
* **echo "obase=2;$no" | bc**
* **1100100**
* **no=1100100**
* **echo "obase=10;ibase=2;$no" | bc**
* **100**
* Calculating exponents and square roots can be done as follows:
* **echo "sqrt(100)" | bc #Square root**

**echo "10^10" | bc #Exponents**

# Playing with file descriptors and redirection

* 0: stdin (standard input)
* 1: stdout (standard output)
* 2: stderr (standard error)
* You can redirect stderr exclusively to a file and stdout to another file as follows:
* **$ cmd 2>stderr.txt 1>stdout.txt**
* It is also possible to redirect stderr and stdout to a single file by converting stderr to stdout using this preferred method:
* **$ cmd 2>&1 output.txt**
* Or the alternate approach:
* **$ cmd &> output.txt**

However, there is a way to redirect data to a file, as well as provide a copy of redirected data as stdin for the next set of commands. This can be done using the tee command. For example, to print stdout in the terminal as well as redirect stdout into a file, the syntax for teeis as follows:

**$ cat a\*.txt | tee out.txt | cat -n**

By default, the tee command overwrites the file, but it can be used with appended options by providing the -a option, for example, $ cat a\* | tee -a out.txt | cat -n.

We can use stdin as a command argument. It can be done by using - as the filename argument for the command as follows:

**$ echo who is this | tee -**

Alternately, we can use /dev/stdin as the output filename to use stdin.

Similarly, use /dev/stderr for standard error and /dev/stdout for standard output.

# Arrays and associative arrays

Define an array:

**array\_var=(1 2 3 4 5 6)**

OR

**array\_var[0]="test1"**

**array\_var[1]="test2"**

Print the contents of an array:

**echo ${array\_var[0]}**

OR

**index=5**

**echo ${array\_var[$index]}**

OR

**$ echo ${array\_var[\*]}**

OR

**$ echo ${array\_var[@]}**

Print the length of an array:

**$ echo ${#array\_var[\*]}**

Define an associative array:

**$ declare -A ass\_array**

Then:

**$ ass\_array=([index1]=val1 [index2]=val2)**

OR

**$ ass\_array[index1]=val1**

**$ ass\_array[index2]=val2**

Obtain the list of indexes in an array:

**$ echo ${!array\_var[\*]}**

OR

**$ echo ${!array\_var[@]}**

# Visiting aliases

An alias can be created as follows:

The alias command is temporary; aliasing exists until we close the current terminal only. To keep these shortcuts permanent, add this statement to the ~/.bashrc file.**$ alias new\_command='command sequence'**

To remove an alias, remove its entry from ~/.bashrc (if any) or use the unalias command.

The \ character escapes the command, running it without any aliased changes.

# Grabbing information about the terminal

* Get the number of columns and rows in a terminal by using the following commands:
* **tput cols**
* **tput lines**
* To print the current terminal name, use the following command:
* **tput longname**
* To move the cursor to a 100,100 position, you can enter:
* **tput cup 100 100**
* To make text bold use this:
* **tput bold**
* To start and end underlining use this:
* **tput smul**
* **tput rmul**

While typing a password, we should not display the characters typed. In the following example, we will see how to do it using stty:

**echo -e "Enter password: "**

**stty –echo**

# Getting and setting dates and delays

use time <scriptpath> to get the time that it took to execute the script

To write a date format to get the output as required, use the following table:

| Date component | Format |
| --- | --- |
| Weekday | %a (for example, Sat)  %A (for example, Saturday) |
| Month | %b (for example, Nov)  %B (for example, November) |
| Day | %d (for example, 31) |
| Date in format (mm/dd/yy) | %D (for example, 10/18/10) |
| Year | %y (for example, 10)  %Y (for example, 2010) |
| Hour | %I or %H (For example, 08) |
| Minute | %M (for example, 33) |
| Second | %S (for example, 10) |
| Nano second | %N (for example, 695208515) |
| Epoch Unix time in seconds | %s (for example, 1290049486) |

# Debugging the script

**$ bash -x script.sh**

Running the script with the -x flag will print each source line with the current status. Note that you can also use sh -x script.

Debug only portions of the script using set -x and set +x. For example:

#!/bin/bash

#Filename: debug.sh

for i in {1..6};

do

set -x

echo $i

set +x

done

echo "Script executed"

(The result of the above totally prints “set +x” in the debug ouput…don’t think it’s supposed to do that.)

* set -x: This displays arguments and commands upon their execution
* set +x: This disables debugging
* set -v: This displays input when they are read
* set +v: This disables printing input

The shebang can be changed from #!/bin/bash to #!/bin/bash -xv to enable debugging without any additional flags

# Functions and arguments

1. A function can be defined as follows:

function fname()

{

statements;

}

Or alternately,

fname()

{

statements;

}

1. A function can be invoked just by using its name:

**$ fname ; # executes function**

1. Arguments can be passed to functions and can be accessed by our script:

**fname arg1 arg2 ; # passing args**

* $1 is the first argument
* $2 is the second argument
* $n is the *n*th argument
* "$@"expands as "$1" "$2" "$3" and so on
* "$\*" expands as "$1c$2c$3", where c is the first character of IFS
* "$@" is used more often than "$\*"since the former provides all arguments as a single string
* **Fork bomb**
* We can write a recursive function, which is basically a function that calls itself:
* :(){ :|:& };:
* A function can be exported—like environment variables—using export, such that the scope of the function can be extended to subprocesses, as follows:
* **export -f fname**
* **echo $?;**
* $? will give the return value of the command cmd.

Passing arguments in many (equivalent) ways:

* $ command -p -v -k 1 file
* $ command -pv -k 1 file
* $ command -vpk 1 file
* $ command file -pvk 1

# Reading the output of a sequence of commands in a variable

In this context, the commands are called **filters** . We connect each filter using pipes, the piping operator being |. An example is as follows:

**$ cmd1 | cmd2 | cmd3**

We can read the output of a sequence of commands combined by pipes as follows:

cmd\_output=$(COMMANDS)

OR

**cmd\_output=`COMMANDS`**

Subshells are separate processes. A subshell can be defined using the ( )operators as follows:

pwd;

(cd /bin; ls);

pwd;

Suppose we are reading the output of a command to a variable using a subshell or the back quotes method. We always quote them in double quotes to preserve the spacing and newline character (\n). For example:

**$ out="$(cat text.txt)"**

# Reading n characters without pressing the return key

The following statement will read *n* characters from input into the variable\_name variable:

**$ read -n 2 var**

Read a password in the nonechoed mode as follows:

**read -s var**

Display a message with read using:

**read -p "Enter input:" var**

Read the input after a timeout as follows:

**$ read -t 2 var**

Use a delimiter character to end the input line as follows:

**$ read -d ":" var**

^^This input does not allow backspace^^

# Running a command until it succeeds

Define a function in the following way:

repeat()

{

while true

do

$@ && return

done

}

# Field separators and iterators

The default value of IFS is a space component (newline, tab, or a space character).

* Using a for loop:
* for var in list;
* do
* commands; # use $var
* done

list can be a string, or a sequence.

We can generate different sequences easily.

echo {1..50}can generate a list of numbers from 1 to 50. echo {a..z}or{A..Z} or {a..h} can generate lists of alphabets. Also, by combining these we can concatenate data.

In the following code, in each iteration, the variable i will hold a character in the range a to z:

for i in {a..z}; do actions; done;

The for loop can also take the format of the for loop in C. For example:

for((i=0;i<10;i++))

{

commands; # Use $i

}

* Using a while loop:
* while condition
* do
* commands;

done

For an infinite loop, use true as the condition.

* Using a until loop:

A special loop called until is available with Bash. This executes the loop until the given condition becomes true. For example:

x=0;

until [ $x -eq 9 ]; # [ $x -eq 9 ] is the condition

do

let x++; echo $x;

done

# Comparisons and tests

* Using an if condition:
* if condition;
* then
* commands;

fi

* Using else if and else:
* if condition;
* then
* commands;
* else if condition; then
* commands;
* else
* commands;

fi

The if conditions can be lengthy, to make them shorter we can use logical operators as follows:

* [ condition ] && action; # action executes if the condition is true
* [ condition ] || action; # action executes if the condition is false

&& is the logical AND operation and || is the logical OR operation. This is a very helpful trick while writing Bash scripts.

Performing mathematical comparisons: Usually conditions are enclosed in square brackets []. Note that there is a space between [ or ] and operands. It will show an error if no space is provided.

* -eq: Equal to
* -ne: Not equal to
* -gt: Greater than
* -lt: Less than
* -ge: Greater than or equal to
* -le: Less than or equal to
* Multiple test conditions can be combined as follows:

[ $var1 -ne 0 -a $var2 -gt 2 ] # using and -a

[ $var1 -ne 0 -o var2 -gt 2 ] # OR -o

* [ -f $file\_var ]: This returns true if the given variable holds a regular file path or filename
* [ -x $var ]: This returns true if the given variable holds a file path or filename that is executable
* [ -d $var ]: This returns true if the given variable holds a directory path or directory name
* [ -e $var ]: This returns true if the given variable holds an existing file
* [ -c $var ]: This returns true if the given variable holds the path of a character device file
* [ -b $var ]: This returns true if the given variable holds the path of a block device file
* [ -w $var ]: This returns true if the given variable holds the path of a file that is writable
* [ -r $var ]: This returns true if the given variable holds the path of a file that is readable
* [ -L $var ]: This returns true if the given variable holds the path of a symlink

String comparisons: While using string comparison, it is best to use double square brackets, since the use of single brackets can sometimes lead to errors.

Two strings can be compared to check whether they are the same in the following manner:

* [[ $str1 = $str2 ]]: This returns true when str1 equals str2, that is, the text contents of str1 and str2 are the same
* [[ $str1 == $str2 ]]: It is an alternative method for string equality check (wtf mate? Why are there so many equivalent ways to write things in the unix shell?)

We can check whether two strings are not the same as follows:

1. [[ $str1 != $str2 ]]: This returns true when str1 and str2 mismatch

We can find out the alphabetically smaller or larger string as follows:

* [[ $str1 > $str2 ]]: This returns true when str1 is alphabetically greater than str2 (alphabetically after str2)
* [[ $str1 < $str2 ]]: This returns true when str1 is alphabetically lesser than str2

### Note

Note that a space is provided after and before =, if it is not provided, it is not a comparison, but it becomes an assignment statement.

* [[ -z $str1 ]]: This returns true if str1 holds an empty string
* [[ -n $str1 ]]: This returns true if str1 holds a nonempty string
* if [ $var -eq 0 ]; then echo "True"; fi
* can be written as
* if test $var -eq 0 ; then echo "True"; fi

# Chapter 2. Have a Good Command

# Concatenating with cat

We can concatenate content from input files along with standard input using cat. Combine stdin and data from another file, as follows:

**$ echo 'Text through stdin' | cat - file.txt**

In this example, - acts as the filename for the stdin text.

If you need to remove the extra blank lines, use the following syntax:

**$ cat -s file**

**----------------**

**$ cat -T file.py**

**def function():**

**^Ivar = 5**

**next = 6**

**^Ithird = 7^I**

**----------------**

**$ cat -n lines.txt**

**1 line**

**2 line**

**3 line**

If you want to skip numbering blank lines, use the -b option.

# Recording and playing back of terminal sessions

**$ script -t 2> timing.log -a output.session**

**$ scriptreplay timing.log output.session**

# Finding files and file listing

**$ find . –print** (all files in current and descending child directories)

**$ find .. –print** (all files in parent and descending child directories)

The –name/-iname argument specifies a matching string for the filename.

**$ find /home/slynux -name "\*.txt" -print**

**$ find . -iname "example\*" –print** (ignores case)

**$ find . \( -name "\*.txt" -o -name "\*.pdf" \) –print** (find TXT or PDF files)

-path matches the file path as a whole. For example:

**$ find /home/users -path "\*/slynux/\*" -print**

The -regex argument is similar to -path, but -regex matches the file paths based on regular expressions.

**$ find . -regex ".\*\(\.py\|\.sh\)$"**

**$ find . -iregex ".\*\(\.py\|\.sh\)$"** (ignores case)

find can also exclude things that match a pattern using !:

**$ find . ! -name "\*.txt" -print**

For specifying the maximum depth we use the -maxdepth level parameter. Similarly, we can also specify the minimum level at which the descending should start.

**$ find . -maxdepth 1 -name "f\*" -print**

**$ find . -mindepth 2 -name "f\*" -print**

-maxdepth and -mindepth should be specified as the third argument to the find command. If they are specified as the fourth or further arguments, it may affect the efficiency

You can use the type arguments from the following table to properly match the required file type:

| File type | Type argument |
| --- | --- |
| Regular file | f |
| Symbolic link | l |
| Directory | d |
| Character special device | c |
| Block device | b |
| Socket | s |
| FIFO | p |

* **Access time** (-atime): It is the last timestamp of when the file was accessed by a user
* **Modification time** (-mtime): It is the last timestamp of when the file content was modified
* **Change time** (-ctime): It is the last timestamp of when the metadata for a file (such as permissions or ownership) was modified
* -amin (access time in min)
* -mmin (modification time in min)
* -cmin (change time in min)
* Print all the files that were accessed within the last seven days as follows:
* **$ find . -type f -atime -7 -print**
* Print all the files that are having access time exactly seven-days old as follows:
* **$ find . -type f -atime 7 -print**
* Print all the files that have an access time older than seven days as follows:

**$ find . -type f -atime +7 -print**

For example, find all the files that have a modification time greater than that of the modification time of a given file.txt file as follows:

**$ find . -type f -newer file.txt -print**

Based on the file sizes of the files, a search can be performed as follows:

**$ find . -type f -size +2k**

**# Files having size greater than 2 kilobytes**

**$ find . -type f -size -2k**

**# Files having size less than 2 kilobytes**

**$ find . -type f -size 2k**

**# Files having size 2 kilobytes**

Instead of k we can use different size units such as the following:

* b: 512 byte blocks
* c: Bytes
* w: Two-byte words
* k: Kilobyte (1024 bytes)
* M: Megabyte (1024 kilobytes)
* G: Gigabyte (1024 megabytes)
* Remove all the .swp files from the current directory as follows:

**$ find . -type f -name "\*.swp" -delete**

It is possible to match files based on the file permissions. We can list out the files having specified file permissions as follows:

**$ find . -type f -perm 644 -print**

**# Print files having permission 644**

For example, to print the list of all files owned by the user slynux, you can use the following command:

**$ find . -type f -user slynux -print**

You must run the find command as root if you want to change ownership of files or directories.

Let's have a look at the following example:

**# find . -type f -user root -exec chown slynux {} \;**

{} is a special string used with the -exec option. For each file match, {} will be replaced with the filename for -exec.

Sometimes we don't want to run the command for each file. Instead, we might want to run it a few times with a list of files as parameters. For this, we use + instead of ; in the exec syntax.

We can use find to match all the C files recursively and use the cat command with the -exec flag as follows:

**$ find . -type f -name "\*.c" -exec cat {} \;>all\_c\_files.txt**

For example, to copy all the .txt files that are older than 10 days to a directory OLD, use the following command:

**$ find . -type f -mtime +10 -name "\*.txt" -exec cp {} OLD \;**

We cannot use multiple commands along with the -exec parameter. It accepts only a single command, but we can use a trick. Write multiple commands in a shell script (for example,commands.sh) and use it with -exec as follows:

-exec ./commands.sh {} \;

-exec can be coupled with printf to produce a very useful output. For example:

**$ find . -type f -name "\*.txt" -exec printf "Text file: %s\n" {} \;**

The technique of excluding files and directories from the search is known as **pruning**. It can be performed as follows:

**$ find devel/source\_path \( -name ".git" -prune \) -o \( -type f -print \)**

# Playing with xargs

When using the pipe operator, the xargs command should always be the first thing to appear after the operator.

**Converting multiple lines of input to a single-line output**:

**$ cat example.txt | xargs**

**1 2 3 4 5 6 7 8 9 10 11 12**

**Converting single-line into multiple-line output**:

**$ cat example.txt | xargs -n 3**

**1 2 3**

**4 5 6**

**7 8 9**

**10 11 12**

To specify a custom delimiter for input, use the -d option as follows:

**$ echo "splitXsplitXsplitXsplit" | xargs -d X**

**split split split split**

For executing a command with X arguments per each execution, use:

INPUT | xargs -n X

**$ cat args.txt | xargs -n 1 ./cecho.sh**

When -I is used with xargs, it will execute as one command execution per argument.

Let's do it as follows:

**$ cat args.txt | xargs -I {} ./cecho.sh -p {} -l**

**-p arg1 -l #**

**-p arg2 -l #**

**-p arg3 -l #**

-I {} specifies the replacement string.

When used with -I, the command is executed in a loop. When there are three arguments the command is executed three times along with the command {}. Each time {} is replaced with arguments one by one.

Let's use find to match and list of all the .txt files and remove them using xargs:

**$ find . -type f -name "\*.txt" -print0 | xargs -0 rm -f**

This is a task most programmers do, that is, counting all C program files for **Lines of Code** (**LOC**). The code for this task is as follows:

**$ find source\_code\_dir\_path -type f -name "\*.c" -print0 | xargs -0 wc -l**

If you want more statistics about your source code, there is a utility called **SLOCCount**, which is very useful. Modern GNU/Linux distributions usually have packages or you can get it from<http://www.dwheeler.com/sloccount/>.

A subshell with a while loop can be used to read arguments and execute commands in a trickier way as follows:

**$ cat files.txt | ( while read arg; do cat $arg; done )**

**# Equivalent to cat files.txt | xargs -I {} cat {}**

# Translating with tr

tr accepts input only through stdin

tr [options] set1 set2

If the length of sets is unequal, set2 is extended to the length of set1 by repeating the last character, or else, if the length of set2 is greater than that of set1, all the characters exceeding the length ofset1 are ignored from set2.

If startchar-endchar is not a valid continuous character sequence, they are then taken as a set of three characters (for example, startchar, -, and endchar). You can also use special characters such as '\t', '\n', or any ASCII characters.

**$ echo "tr came, tr saw, tr conquered." | tr 'a-zA-Z' 'n-za-mN-ZA-M'**

**ge pnzr, ge fnj, ge pbadhrerq.**

**$ echo "Hello 123 world 456" | tr -d '0-9'**

**Hello world**

**$ echo hello 1 char 2 next 4 | tr -d -c '0-9 \n'**

**1 2 4**

**$ echo "GNU is not UNIX. Recursive right ?" | tr -s ' '**

**GNU is not UNIX. Recursive right ?**

**$ cat sum.txt | echo $[ $(tr '\n' '+' ) 0 ]**

**15**

$[ operation ] performs a numeric operation. Hence, it forms the string as follows:

**echo $[ 1+2+3+4+5+0 ]**

tr can use different character classes as sets. The different classes are as follows:

* alnum: Alphanumeric characters
* alpha: Alphabetic characters
* cntrl: Control (nonprinting) characters
* digit: Numeric characters
* graph: Graphic characters
* lower: Lowercase alphabetic characters
* print: Printable characters
* punct: Punctuation characters
* space: Whitespace characters
* upper: Uppercase characters
* xdigit: Hexadecimal characters

For example:

* **tr '[:lower:]' '[:upper:]'**

# Checksum and verification

The syntax for the md5sum checksum calculation is as follows:

**$ md5sum file1 file2 file3 ..**

The integrity of a file can be verified by using the generated file as follows:

**$ md5sum -c file\_sum.md5**

**$ md5deep -rl directory\_path > directory.md5**

**# -r to enable recursive traversal**

**# -l for using relative path. By default it writes absolute file path in output**

Alternately, use a combination of find to calculate checksums recursively:

**$ find directory\_path -type f -print0 | xargs -0 md5sum >> directory.md5**

# Cryptographic tools and hashes

**$ crypt PASSPHRASE <input\_file >encrypted\_file**

In order to encrypt a file with gpg use:

**$ gpg -c filename**

This command reads the passphrase interactively and generates filename.gpg. In order to decrypt a gpg file use:

**$ gpg filename.gpg**

In order to encode a binary file into the Base64 format, use:

**$ base64 filename > outputfile**

Decode Base64 data as follows:

**$ base64 -d file > outputfile**

**md5sum** and **SHA-1** are unidirectional hash algorithms. They are no longer secure.

It is recommended to use tools such as bcrypt or sha512sum instead. Read more about this at <http://codahale.com/how-to-safely-store-a-password/>

Shadow passwords are usually salted passwords. SALT is an extra string used to obfuscate and make the encryption stronger. The salt consists of random bits that are used as one of the inputs to a key derivation function that generates the salted hash for the password.

**$ openssl passwd -1 -salt SALT\_STRING PASSWORD**

**$1$SALT\_STRING$323VkWkSLHuhbt1zkSsUG.**

# Sorting unique and duplicates

The sort command accepts input as filenames, as well as from stdin (standard input) and outputs the result by writing into stdout. The same applies to the uniq command.

1. We can easily sort a given set of files (for example, file1.txt and file2.txt) as follows:

**$ sort file1.txt file2.txt > sorted.txt**

1. For a numerical sort, we can use:

**$ sort -n file.txt**

1. To sort in the reverse order, we can use:

**$ sort -r file.txt**

1. For sorting by months (in the order Jan, Feb, March,…), use:

**$ sort -M months.txt**

1. To merge two already sorted files, use:

**$ sort -m sorted1 sorted2**

1. To find the unique lines from a sorted file, use:

**$ sort file1.txt file2.txt | uniq**

sort returns an exit code ($?) of 0 if the file is sorted and nonzero otherwise

**# Sort by column 2**

**$ sort -k 2 data.txt**

**# Sort numerically/reverse by column1**

**$ sort -nrk 1 data.txt**

To use the first character as the key, use:

**$ sort -nk 1,1 data.txt**

To make the sort's output xargs compatible with the \0 terminator, use the following command:

**$ sort -z data.txt | xargs -0**

To sort them in dictionary order, by ignoring punctuations and folds, use:

**$ sort -bd unsorted.txt**

**$ uniq sorted.txt**

OR

**$ sort unsorted.txt | uniq**

Display only unique lines (the lines which are not repeated or duplicated in the input file) as follows:

**$ uniq -u sorted.txt**

OR

**$ sort unsorted.txt | uniq -u**

To count how many times each of the lines appears in the file, use the following command:

**$ sort unsorted.txt | uniq -c**

* -s specifies the number for the first *N* characters to be skipped
* -w specifies the maximum number of characters to be compared

**$ sort data.txt | uniq -s 2 -w 2 (ignore first two, compare next two)**

Zero-byte-terminated output can be generated from the uniq command as follows:

**$ uniq -z file.txt**

The following command removes all the files, with filenames read from file.txt:

**$ uniq -z file.txt | xargs -0 rm**

# Temporary file naming and random numbers

Create a temporary file as follows:

**$ filename=`mktemp`**

To create a temporary directory, use the following commands:

**$ dirname=`mktemp -d`**

To just generate a filename without actually creating a file or directory, use this:

**$ tmpfile=`mktemp -u`**

To create the temporary filename according to a template, use:

**$mktemp test.XXX**

X will be replaced by a random alphanumeric character. Also note that there must be at least three X characters in the template for mktemp to work.

# Splitting files and data

You can split this file into smaller files of 10k each by specifying the split size as follows:

**$ split -b 10k data.file**

To use the numeric suffixes, use an additional -d argument. It is also possible to specify a suffix length using -a length:

**$ split -b 10k data.file -d -a 4**

Instead of the k (kilobyte) suffix we can use M for MB, G for GB, c for byte, w for word

**$ split -b 10k data.file -d -a 4 split\_file**

**$ ls**

**data.file split\_file0002 split\_file0005 split\_file0008 strtok.c**

**split\_file0000 split\_file0003 split\_file0006 split\_file0009**

**split\_file0001 split\_file0004 split\_file0007**

To split files based on the number of lines in each split rather than chunk size, use -l no\_of\_lines as follows:

**$ split -l 10 data.file**

csplit makes the split based on context. It can be used to split files based on the existence of a certain word or text content.

We may need to split the files into server1.log, server2.log, and server3.log from the contents for each SERVER in each file. This can be done as follows:

**$ csplit server.log /SERVER/ -n 2 -s {\*} -f server -b "%02d.log" ; rm server00.log**

* /SERVER/ is the line used to match a line by which a split is to be carried out.
* /[REGEX]/ is the format. It copies from the current line (first line) up to the matching line that contains "SERVER" excluding the match line.
* {\*} is used to specify to repeat a split based on the match up to the end of the file. By using {integer}, we can specify the number of times it is to be continued.
* -s is the flag to make the command silent rather than printing other messages.
* -n is used to specify the number of digits to be used as suffix. 01, 02, 03, and so on.
* -f is used for specifying the filename prefix for split files (server is the prefix in the previous example).
* -b is used to specify the suffix format. "\_d.log" is similar to the printf argument format in C. Here, the filename = prefix + suffix, that is, "server" + "\_d.log".

# Slicing filenames based on extension

The name from name.extension can be easily extracted using the % operator. You can extract the name from "sample.jpg" as follows:

file\_jpg="sample.jpg"

name=${file\_jpg%.\*}

echo File name is: $name

Extract .jpg from the filename stored in the variable file\_jpg as follows:

extension=${file\_jpg#\*.}

echo Extension is: $extension

% is a nongreedy operation. It finds the minimal match for the wildcard from right to left. There is an operator %%, which is similar to %. But it is greedy in nature. This means, it finds the maximal match of the string for the wildcard.

In the second task, we have used the # operator to extract the extension from the filename. It is similar to %. But it evaluates from left to right.

Similarly, as in the case of %%, we have another greedy operator for #, which is ##.

The ## operator is preferred over the # operator to extract the extension from a filename since the filename may contain multiple "." characters. Since ## makes a greedy match, it always extracts extensions only.

# Renaming and moving files in bulk

* Renaming \*.JPG to \*.jpg:
* **$ rename \*.JPG \*.jpg <THIS DOESN’T WORK?!**
* To replace space in the filenames with the "\_" character:
* **$ rename 's/ /\_/g' \***
* To convert any filename from uppercase to lowercase and vice versa:
* **$ rename 'y/A-Z/a-z/' \***
* **$ rename 'y/a-z/A-Z/' \***
* To recursively move all the .mp3 files to a given directory:
* **$ find path -type f -name "\*.mp3" -exec mv {} target\_dir \;**
* To recursively rename all files by replacing space with the "\_" character:

**$ find path -type f -exec rename 's/ /\_/g' {} \;**

# Spell checking and dictionary manipulation

The /usr/share/dict/ directory contains some of the dictionary files.

In grep, ^ is the word-start-marker character and the $ character is the word-end marker. -q is used to suppress any output and to be silent.

The aspell list command returns output text when the given input is not a dictionary word, and does not output anything when the input is a dictionary word. A -z command checks whether a string is empty or not.

List all words in a file starting with a given word as follows:

**$ look word filepath**

Or alternately, use:

**$ grep "^word" filepath**

# Automating interactive input

#!/bin/bash

#Filename: interactive.sh

read -p "Enter number:" no ;

read -p "Enter name:" name

echo You have entered $no, $name;

Let's automate the sending of input to the command as follows:

**$ echo -e "1\nBobby\n" | ./interactive.sh**

**You have entered 1, Bobby**

You can also manually craft the input file without the echo commands by hand typing. For example:

**$ ./interactive.sh < input.data**

In order to handle a dynamic input supply and provide input by checking the input requirements by the program on runtime, we have a great utility called expect. The expect command supplies the correct input for the correct input prompt by the program.

#!/usr/bin/expect

#Filename: automate\_expect.sh

spawn ./interactive .sh

expect "Enter number:"

send "1\n"

expect "Enter name:"

send "hello\n"

expect eof

# Making commands quicker by running parallel processes

#/bin/bash

#filename: generate\_checksums.sh

PIDARRAY=()

for file in File1.iso File2.iso

do

md5sum $file &

PIDARRAY+=("$!")

done

wait ${PIDARRAY[@]}

We exploit the Bash operand &, which instructs the shell to send the command to the background and continue with the script. However, this means that our script will exit as soon as the loop completes while the md5sum processes are still running in the background. To prevent this, we get the PIDs of the processes using $!, which in Bash holds the PID of the last background process. We append these PIDs to an array and then use the wait command to wait for these processes to finish.

**Chapter 3. File In, File Out**

# Introduction

Unix treats every object in the operating system as a file.

# Generating files of any size

**loopback files** are files that can contain a filesystem itself and these files can be mounted similarly to a physical device using the mount command

The dd command clones the given input and writes an exact copy to the output.

**$ dd if=/dev/zero of=junk.data bs=1M count=1**

Be careful while using the dd command, it operates on a very low level with the devices. If you make a mistake, you might end up wiping your disk or corrupting data otherwise. So, always double check your dd command syntax, especially your of= parameter for correctness.

Append any of the following characters to the number to specify the size in bytes:

| Unit size | Code |
| --- | --- |
| Byte (1 B) | c |
| Word (2 B) | w |
| Block (512 B) | b |
| Kilobyte (1024 B) | k |
| Megabyte (1024 KB) | M |
| Gigabyte (1024 MB) | G |

If the input parameter (if) is not specified, it will read the input from stdin by default. Similarly, if the output parameter (of) is not specified, it will use stdout as the default output sink.

# The intersection and set difference (A-B) on text files

comm takes only sorted files as input.

First, execute comm without any options:

**$ comm A.txt B.txt**

**apple**

**carrot**

**cookies**

**gold**

**iron**

**orange**

**silver**

**steel**

Column 1 = Only A

Column 2 = Only B

Column 3 = Intersection

In order to print the intersection of two files, we need to remove the first and second columns and print the third column only as follows:

**$ comm A.txt B.txt -1 -2**

Print lines that are uncommon in two files as follows:

**$ comm A.txt B.txt -3**

In order to produce a unified output, use the following command line:

**$ comm A.txt B.txt -3 | sed 's/^\t//'**

Set difference for A.txt:

**$ comm A.txt B.txt -2 -3**

s in the sed script stands for substitute.

# Working with file permissions, ownership, and the sticky bit

Permissions of a file can be listed by using the ls -l command:

**-rw-r--r-- 1 slynux slynux 2497 2010-02-28 11:22 bot.py**

The first column of the output specifies the following, with the first letter corresponding to:

* - – if it is a regular file
* d – if it is a directory
* c – for a character device
* b – for a block device
* l – if it is a symbolic link
* s – for a socket
* p – for a pipe

The rest can be divided into three groups of three letters each (--- --- ---). The first --- three characters correspond to the permissions of the user (owner), the second set of three characters correspond to the permissions of the group, and the third set of three characters correspond to the permissions of others.

* **User** (permission string: rwx------): The first letter in the three letters specifies whether the user has read permission for the file. If the read permission is set for the user, the character r will appear as the first character. Similarly, the second character specifies write (modify) permission (w) and the third character specifies whether the user has execute (x) permission (the permission to run the file). The execute permission is usually set for executable files. The user has one more special permission called **setuid** (S), which appears in the position of execute (x). The setuid permission enables an executable file to be executed effectively as its owner, even when the executable is run by another user.

An example for a file with setuid permission set is -rwS------.

The read, write, and execute permissions are also applied to the directories. However, the meanings of read, write, and execute permissions are slightly different in the context of directories as follows:

* + The read permission (r) for the directories enables reading the list of files and subdirectories in the directory
  + The write permission (w) for a directory enables creating or removing files and directories from a directory
  + The execute permission (x) specifies whether the access to the files and directories in a directory is possible or not
* **Group** (permission string: ---rwx---): Instead of setuid, the group has a **setgid** (S) bit. This enables the item to run an executable file with an effective group as the owner group.
* **Others** (permission string: ------rwx): It does not have permission S (such as setuid or setgid).

Directories have a special permission called a **sticky bit** . When a sticky bit is set for a directory, only the user who created the directory can delete the files in the directory, even if the group and others have write permissions. The sticky bit appears in the position of execute character (x) in the others permission set. It is represented as character t or T. The t character appears in the position of x if the execute permission is unset and the sticky bit is set. If the sticky bit and the execute permission are set, the character T appears in the position of x.

This could be set using chmod as follows:

**$ chmod u=rwx g=rw o=r filename**

Here:

* u – specifies user permissions
* g – specifies group permissions
* o – specifies others permissions

Use + to add permission to a user, group, or others and use - to remove the permissions.

**$ chmod o+x filename**

**$ chmod a+x filename #a means all**

**$ chmod a-x filename**

Read, write, and execute permissions have unique octal numbers as follows:

* r-- = 4
* -w- = 2
* --x = 1
* rwx = 4 + 2 + 1 = 7
* rw- = 4 + 2 = 6
* r-- = 4

Therefore, rwx rw- r-- is equal to 764, and the command for setting the permissions using octal values is:

* **$ chmod 764 filename**

In order to change ownership of files, use the chown command as follows:

* **$ chown user.group filename**

Sometimes it may be required to recursively change the permissions of all the files and directories inside the current directory. This can be done as follows:

* **$ chmod 777 . -R**

We can apply the ownership recursively by using the -R flag with the chown command as follows:

* **$ chown user.group . -R**

First, change the ownership to the user that needs to execute it and then log in as the user. Then, run the following command:

**$ chmod +s executable\_file**

**# chown root.root executable\_file**

**# chmod +s executable\_file**

**$ ./executable\_file**

Now it executes effectively as the root user every time. However, setuid is restricted such that setuid won't work for scripts.

# Making files immutable

1. A file can be made immutable using the following command:

**# chattr +i file**

1. In order to make it writable again, remove the immutable attribute as follows:

**chattr -i file**

# Generating blank files in bulk

1. A blank file with the name filename will be created using the following command:

**$ touch filename**

Generate bulk files with a different name pattern as follows:

for name in {1..100}.txt

do

touch $name

done

If a file already exists, the touch command changes all timestamps associated with the file to the current time. However, if we want to specify that only certain stamps are to be modified, we use the following options:

* touch -a modifies only the access time
* touch -m modifies only the modification time

Instead of using the current time for the timestamp, we can specify the time and date with which to stamp the file as follows:

**$ touch -d "Fri Jun 25 20:50:14 IST 1999" filename**

# Finding symbolic links and their targets

Symbolic links are just pointers to other files, they are similar in function to aliases in Mac OS X or shortcuts in Windows.

We can create a symbolic link as follows:

**$ ln -l -s /var/www/ ~/web #~/web is the alias**

To verify that the link was created, run the following command:

**$ ls -l web**

**lrwxrwxrwx 1 slynux slynux 8 2010-06-25 21:34 web -> /var/www**

1. In order to print symbolic links in the current directory, use the following command:

**$ ls -l | grep "^l"**

1. Use find to print all symbolic links from the current directory and subdirectories:

**$ find . -type l -print**

To read the target path for a given symbolic link, use the readlink command:

**$ readlink web**

# Enumerating file type statistics

 In Unix/Linux systems, file types are not determined based on the extension of the file (like the Microsoft Windows platform does).

To print the type of a file use the following command:

**$ file filename**

Print the file type only by excluding the filename as follows:

**$ file -b filename**

# Creating ISO files and hybrid ISO

The important thing to note here is that just copying files from a bootable CD-ROM to another one is not sufficient to make the new one bootable. To preserve the bootable nature of a CD-ROM, it should be copied as a disk image using an ISO file.

The preferred way to create an ISO image is to use dd:

**# dd if=/dev/cdrom of=image.iso**

We can use mkisofs to create an ISO file using a directory containing all the required files that should appear as contents of an ISO file as follows:

**$ mkisofs -V "Label" -o image.iso source\_dir/**

The -o option in the mkisofs command specifies the ISO file path. The source\_dir command is the path of the directory that should be used as source content for the ISO and the -Voption specifies the label that should be used for the ISO file.

We can convert standard ISO files into hybrid ISOs with the following command:

**# isohybrid image.iso**

To write the ISO to a USB storage device, use the following command:

**# dd if=image.iso of=/dev/sdb1**

or, you can use cat as follows:

**# cat image.iso >> /dev/sdb1**

The cdrecord command is used to burn an ISO file into a CD-ROM or DVD-ROM. It can be used to burn the image to the CD-ROM as follows:

**# cdrecord -v dev=/dev/cdrom image.iso**

Some extra options are as follows:

* We can specify the burning speed with the -speed option as follows:

**-speed SPEED (4,8,16,32,…)**

Multisession burning can be performed using the -multi option as follows:

**# cdrecord -v dev=/dev/cdrom image.iso -multi**

If you are on a desktop computer, try the following commands and have fun:

**$ eject**

This command is used to eject the tray.

**$ eject -t**

This command is used to close the tray.

# Finding the difference between files, patching

Sending a difference file instead is very helpful as it consists of only lines which are changed, or added or removed and line numbers are attached with it. This difference file is called a **patch file** .

1. Nonunified diff output (without the -u flag) will be as follows:

**$ diff version1.txt version2.txt**

**3d2**

**<line3**

**6c5**

**> GNU is not UNIX**

The unified diff output will be as follows:

**$ diff -u version1.txt version2.txt**

**--- version1.txt 2010-06-27 10:26:54.384884455 +0530**

**+++ version2.txt 2010-06-27 10:27:28.782140889 +0530**

**@@ -1,5 +1,5 @@**

**this is the original text**

**line2**

**-line3**

**line4**

**happy hacking !**

**-**

**+GNU is not UNIX**

A patch file can be generated by redirecting the diff output to a file, as follows:

**$ diff -u version1.txt version2.txt > version.patch**

To apply the patch, use the following command:

**$ patch -p1 version1.txt < version.patch**

To revert the changes back, use the following command:

**$ patch -p1 version1.txt < version.patch**

The diff command can also act recursively against directories. It will generate a difference output for all the descendant files in the directories. Use the following command:

**$ diff -Naur directory1 directory2**

The interpretation of each of the previous options is as follows:

* -N is for treating absent files as empty
* -a is to consider all files as text files
* -u is to produce unified output
* -r is to recursively traverse through the files in the directories

# Using head and tail for printing the last or first 10 lines

1. Print the first 10 lines as follows:

**$ head file**

1. Read the data from stdin as follows:

**$ cat text | head**

1. Specify the number of first lines to be printed as follows:

**$ head -n 4 file**

This command prints four lines.

1. Print all lines excluding the last M lines as follows:

**$ head -n -M file**

1. Printing by excluding the last lines is a very important usage of head. Now, let us see how to print, last few lines. Print the last 10 lines of a file as follows:

**$ tail file**

1. In order to read from stdin, you can use the following command line:

**$ cat text | tail**

1. Print the last five lines as follows:

**$ tail -n 5 file**

1. In order to print all lines excluding the first M lines, use the following code:

**$ tail -n +(M+1)**

One of the important usages of tail is to read a constantly growing file. To constantly monitor the growth of file, tail has a special option -f or –follow.

You will probably want to use this on logfiles. The command to monitor the growth of the files would be:

**# tail -f /var/log/messages**

Or:

**$ dmesg | tail -f**

The -f tail can also add a sleep interval -s, so that we can set the interval during which the file updates are monitored.

tail has the interesting property that allows it to terminate after a given process ID dies.

Suppose we are reading a growing file, and a process Foo is appending data to the file, the -f tail should be executed until the process Foo dies.

**$ PID=$(pidof Foo)**

**$ tail -f file --pid $PID**

# Listing only directories – alternative methods

There are several ways in which directories in the current path can be displayed:

1. Using ls with -d to print directories:

**$ ls -d \*/**

1. Using ls -F with grep:

**$ ls -F | grep "/$"**

1. Using ls -l with grep:

**$ ls -l | grep "^d"**

1. Using find to print directories:

**$ find . -type d -maxdepth 1 -print**

When the -F parameter is used with ls, all entries are appended with some type of file characters such as @, \*, |, and so on. For directories, entries are appended with the / character.

# Fast command-line navigation using pushd and popd

pushd and popd operate on a stack. We know that a stack is a **last** **infirst** **out** (**LIFO**) data structure.

1. To push and change a directory to a path, use the following command:

**~ $ pushd /var/www**

Now the stack contains /var/www ~ and the current directory is changed to /var/www.

1. Now, again push the next directory path as follows:

**/var/www $ pushd /usr/src**

Now the stack contains /usr/src /var/www ~ and the current directory is /usr/src.

You can similarly push as many directory paths as needed.

1. To view the stack contents, use the following command:

**$ dirs**

**/usr/src /var/www ~ /usr/share /etc**

**0 1 2 3 4**

1. Now when you want to switch to any path in the list, number each path from 0 to n, then use the path number for which we need to switch, for example:

**$ pushd +3**

Now it will rotate the stack and switch to the /usr/share directory.

pushd will always add paths to the stack, to remove paths from the stack use popd.

1. To remove a last pushed path and change directory to the next directory, use the following command:

**$ popd**

Suppose the stack is /usr/src /var/www ~ /usr/share /etc such that the current directory is /usr/src, then popd will change the stack to /var/www ~ /usr/share /etc and change the directory to /var/www.

1. To remove a specific path from the list, use popd +num:

num is counted as 0 to n from left to right.

When you use only two locations, there is an alternative and easier way. That is cd –

# Counting the number of lines, words, and characters in a file

1. Count the number of lines in the following manner:

**$ wc -l file**

1. To use stdin as input, use the following command:

**$ cat file | wc -l**

1. Count the number of words as follows:

**$ wc -w file**

**$ cat file | wc -w**

1. In order to count the number of characters, use the following commands:

**$ wc -c file**

**$ cat file | wc -c**

For example, we can count the characters in a text as follows:

**echo -n 1234 | wc -c**

**4**

-n is used to avoid an extra newline character.

1. To print the number of lines, words, and characters, execute wc without any options:

**$ wc file**

**1435 15763 112200**

Those are the number of lines, words, and characters respectively.

1. Print the length of the longest line in a file using the -L option:

**$ wc file -L**

**205**

# Printing the directory tree

The following is a sample Unix filesystem tree to show an example:

**$ tree ~/unixfs**

To highlight only files matched by the pattern, use the following syntax:

**$ tree ~/unixfs -P "\*.txt" <-only .txt files**

To highlight only files excluding the match pattern:

**$ tree path -I "\*.txt" <-everything but .txt files**

To print the size along with files and directories, use the -h option:

**$ tree -h**

use the following command to create an HTML file with the tree output:

**$ tree PATH -H http://localhost -o out.html**

# Chapter 4. Texting and Driving

| regex | Description | Example |
| --- | --- | --- |
| ^ | This specifies the start of the line marker. | ^tux matches a line that starts with tux. |
| $ | This specifies the end of the line marker. | tux$ matches a line that ends with tux. |
| . | This matches any one character. | Hack. matches Hack1, Hacki, but not Hack12 or Hackil; only one additional character matches. |
| [] | This matches any one of the characters enclosed in [chars]. | coo[kl] matches cook or cool. |
| [^] | This matches any one of the characters except those that are enclosed in [^chars]. | 9[^01] matches 92 and 93, but not 91 and 90. |
| [-] | This matches any character within the range specified in []. | [1-5] matches any digits from 1 to 5. |
| ? | This means that the preceding item must match one or zero times. | colou?r matches color or colour, but not colouur. |
| + | This means that the preceding item must match one or more times. | Rollno-9+ matches Rollno-99 and Rollno-9, but not Rollno-. |
| \* | This means that the preceding item must match zero or more times. | co\*l matches cl, col, and coool. |
| () | This treats the terms enclosed as one entity | ma(tri)?x matches max or matrix. |
| {n} | This means that the preceding item must match n times. | [0-9]{3} matches any three-digit number. [0-9]{3} can be expanded as [0-9][0-9][0-9]. |
| {n,} | This specifies the minimum number of times the preceding item should match. | [0-9]{2,} matches any number that is two digits or longer. |
| {n, m} | This specifies the minimum and maximum number of times the preceding item should match. | [0-9]{2,5} matches any number that has two digits to five digits. |
| | | This specifies the alternation—one of the items on either of side of| should match. | Oct (1st | 2nd) matches Oct 1st or Oct 2nd. |
| \ | This is the escape character for escaping any of the special characters mentioned previously. | a\.b matches a.b, but not ajb. It ignores the special meaning of .because of \. |

<http://www.linuxforu.com/2011/04/sed-explained-part-1/>

^^MORE DETAILS THAN TABLE ABOVE^^

<http://www.regexper.com/>

^^COOL VISUALIZER OF REGEX^^

# Searching and mining a text inside a file with grep

To search for lines of text that contain the given pattern:

**$ grep pattern filename**

Or:

**$ grep "pattern" filename**

We can also read from stdin as follows:

**$ echo -e "this is a word\nnext line" | grep word**

Perform a search in multiple files by using a single grep invocation, as follows:

**$ grep "match\_text" file1 file2 file3**

We can highlight the word in the line by using the --color option as follows:

**$ grep word filename --color=auto**

To use the full set of regular expressions as input arguments, the -E option should be added, which means an extended regular expression. Or, we can use an extended regular expression enabled grep command, egrep. For example:

**$ grep -E "[a-z]+" filename**

Or:

**$ egrep "[a-z]+" filename**

In order to output only the matching portion of a text in a file, use the -o option as follows:

**$ echo this is a line. | egrep -o "[a-z]+\."**

In order to print all of the lines, except the line containing match\_pattern, use:

**$ grep -v match\_pattern file**

Count the number of lines in which a matching string or regex match appears in a file or text, as follows:

**$ grep -c "text" filename**

To count the number of matching items in a file, use the following trick:

**$ echo -e "1 2 3 4\nhello\n5 6" | egrep -o "[0-9]" | wc -l**

Print the line number of the match string as follows:

**$ grep “pattern” -n file.txt**

or

**$ cat file.txt | grep “pattern” –n**

Print the character or byte offset at which a pattern matches, as follows:

**$ echo gnu is not unix | grep -b -o "not"**

**7:not**

To search over multiple files, and list which files contain the pattern, we use the following:

**$ grep -l linux sample1.txt sample2.txt**

The inverse of the -l argument is -L.

To recursively search for a text over many directories of descendants, use the following command:

**$ grep "text" . -R -n**

The options -R and -r mean the same thing when used with grep.

The -i argument helps match patterns to be evaluated, without considering the uppercase or lowercase. For example:

**$ echo hello world | grep -i "HELLO"**

However, we can use an argument -e to specify multiple patterns for matching, as follows:

**$ grep -e "pattern1" -e "pattern"**

We can use a pattern file for reading patterns. Write patterns to match line-by-line, and execute grep with a -f argument as follows:

**$ grep -f pattern\_filesource\_filename**

To search only for .c and .cpp files recursively in a directory by excluding all other file types, use the following command:

**$ grep "main()" . -r --include \*.{c,cpp}**

Exclude all README files in the search, as follows:

**$ grep "main()" . -r --exclude "README"**

To exclude directories, use the --exclude-dir option.

To read a list of files to exclude from a file, use --exclude-from FILE.

In the following command sequence, grep outputs filenames with a zero-byte terminator (\0), because of the -Z option with grep. xargs -0reads the input and separates filenames with a zero-byte terminator:

**$ grep "test" file\* -lZ | xargs -0 rm**

Usually, -Z is used along with -l.

Sometimes, instead of actually looking at the matched strings, we are only interested in whether there was a match or not. For this, we can use the quiet option (-q), where the grep command does not write any output to the standard output. Instead, it runs the command and returns an exit status based on success or failure.

In order to print three lines after a match, use the -A option:

**$ seq 10 | grep 5 -A 3**

In order to print three lines before the match, use the -B option:

**$ seq 10 | grep 5 -B 3**

Print three lines after and before the match, and use the -C option as follows:

**$ seq 10 | grep 5 -C 3**

If there are multiple matches, then each section is delimited by a line "--":

# Cutting a file column-wise with cut

To extract particular fields or columns, use the following syntax:

**$ cut -f 2,3 filename**

Extract multiple fields as follows:

**$ cut -f2-4 student\_data.txt <- space unnecessary**

Suppose you have many fields and you want to print all the columns except the third column, then use the following command:

**$ cut -f3 --complement student\_data.txt**

**No Name Percent**

**1 Sarath 90**

**2 Alex 98**

**3 Anu 90**

To specify the delimiter character for the fields, use the -d option as follows:

**$ cut -f2 -d";" delimited\_data.txt**

|  |  |
| --- | --- |
| N- | from the Nth byte, character, or field, to the end of the line |
| N-M | from the Nth to Mth (included) byte, character, or field |
| -M | from the first to Mth (included) byte, character, or field |

We use the preceding notations to specify fields as a range of bytes or characters with the following options:

* -b for bytes
* -c for characters
* -f for defining fields

For example:

You can print the first to fifth characters as follows:

**$ cut -c1-5 range\_fields.txt**

We can specify the output delimiter while using with -c, -f, and -b, as follows:

--output-delimiter "delimiter string"

# Using sed to perform text replacement

sed can be matched using regular expressions.

1. **$ sed 's/pattern/replace\_string/' file**

Or:

**$ cat file | sed 's/pattern/replace\_string/'**

If you use the vi editor, you will notice that the command to replace the text is very similar to the one discussed here.

By default, sed only prints the substituted text. To save the changes along with the substitutions to the same file, use the -i option.

**$ sed -i 's/text/replace/' file**

These usages of the sed command will replace the first occurrence of the pattern in each line. If we want to replace every occurrence, we need to add the g parameter at the end, as follows:

**$ sed 's/pattern/replace\_string/g' file**

The /g suffix means that it will substitute every occurrence. However, we sometimes need to replace only the Nth occurrence onwards. For this, we can use the /Ng form of the option.

**$ echo thisthisthisthis | sed 's/this/THIS/2g'**

**thisTHISTHISTHIS**

We can use any delimiter characters as follows:

**sed 's:text:replace:g'**

**sed 's|text|replace|g'**

When the delimiter character appears inside the pattern, we have to escape it using the \ prefix, as follows:

**sed 's|te\|xt|replace|g'**

Removing blank lines is a simple technique by using sed to remove blank lines. Blanks can be matched with regular expression ^$:

**$ sed '/^$/d' file**

/pattern/d will remove lines matching the pattern.

**$ sed -i 's/\b[0-9]\{3\}\b/NUMBER/g' sed\_data.txt**

The preceding one-liner replaces three-digit numbers only. \b[0-9]\{3\}\b is the regular expression used to match three-digit numbers. [0-9]is the range of digits; that is, from 0 to 9. {3} is used for matching the preceding character thrice. \ in \{3\} is used to give a special meaning for{ and }. \b is the word boundary marker.

It's a useful practice to first try the sed command without -i to make sure your regex is correct, and once you are satisfied with the result, add the -i option to actually make changes to the file.

In sed, we can use & as the matched string for the substitution pattern:

**$ echo this is an example | sed 's/\w\+/[&]/g'**

We can also match the substrings of the given pattern. Let's see how to do it.

**$ echo this is digit 7 in a number | sed 's/digit \([0-9]\)/\1/'**

For the first substring match, the corresponding notation is \1; for the second, it is \2, and so on.

**$ echo seven EIGHT | sed 's/\([a-z]\+\) \([A-Z]\+\)/\2 \1/'**

([a-z]\+\) matches the first word, and \([A-Z]\+\) matches the second word. \1 and \2 are used for referencing them. This type of referencing is called **back referencing**. In the replacement part, their order is changed as \2 \1 and, hence, it appears in reverse order.

The combination of multiple sed using a pipe can be replaced as follows:

**sed 'expression' | sed 'expression'**

The preceding command is equivalent to the following:

**$ sed 'expression; expression'**

Or:

**$ sed -e 'expression' -e expression'**

Double quotes expand the expression by evaluating it. Using double quotes are useful when we want to use a variable string in a sed expression.

For example:

**$ text=hello**

**$ echo hello world | sed "s/$text/HELLO/"**

# Using awk for advanced text processing

The structure of an awk script is as follows:

**awk ' BEGIN{ print "start" } pattern { commands } END{ print "end" } file**

Let's go through the following example:

**$ echo -e "line1\nline2" | awk 'BEGIN{ print "Start" } { print } END{ print "End" } '**

**Start**

**line1**

**line2**

**End**

When print is used without an argument, it will print the current line. There are two important things to be kept in mind about it. When the arguments of the print are separated by commas, they are printed with a space delimiter. Double quotes are used as the concatenation operator in the context of print in awk.

For example:

**$ echo | awk '{ var1="v1"; var2="v2"; var3="v3"; \**

**print var1,var2,var3 ; }'**

The preceding statement will print the values of the variables as follows:

**v1 v2 v3**

The echo command writes a single line into the standard output. Hence, the statements in the { } block of awk are executed once. If the standard input to awk contains multiple lines, the commands in awk will be executed multiple times.

Some special variables that can be used with awk are as follows:

* NR: It stands for the current record number, which corresponds to the current line number when it uses lines as records
* NF: It stands for the number of fields, and corresponds to the number of fields in the current record under execution (fields are delimited by space)
* $0: It is a variable that contains the text content of the current line under execution
* $1: It is a variable that holds the text of the first field
* $2: It is the variable that holds the text of the second field

awk also provides the printf() function with the same syntax as in C. We can also use that instead of print.

Print the second and third field of every line as follows:

**$awk '{ print $3,$2 }' file**

In order to count the number of lines in a file, use the following command:

**$ awk 'END{ print NR }' file**

You can sum up all the numbers from each line of field 1 as follows:

**$ seq 5 | awk 'BEGIN{ sum=0; print "Summation:" }**

**{ print $1"+"; sum+=$1 } END { print "=="; print sum }'**

There is a flexible alternate method to pass many variable values from outside awk. For example:

**$ var1="Variable1" ; var2="Variable2"**

**$ echo | awk '{ print v1,v2 }' v1=$var1 v2=$var2**

When an input is given through a file rather than standard input, use the following command:

**$ awk '{ print v1,v2 }' v1=$var1 v2=$var2 filename**

The syntax is getline var. The variable var will contain the content for the line. If getline is called without an argument, we can access the content of the line by using $0, $1, and $2.

For example:

**$ seq 5 | awk 'BEGIN { getline; print "Read ahead first line", $0 } { print $0 }'**

We can specify some conditions for lines to be processed. For example:

**$ awk 'NR < 5' # first four lines**

**$ awk 'NR==1,NR==4' #First four lines**

**$ awk '/linux/' # Lines containing the pattern linux (we can specify regex)**

**$ awk '!/linux/' # Lines not containing the pattern linux**

By default, the delimiter for fields is a space.  We can explicitly specify a delimiter by using -F "delimiter":

**$ awk -F: '{ print $NF }' /etc/passwd**

Or:

**awk 'BEGIN { FS=":" } { print $NF }' /etc/passwd**

We can set the output fields separator by setting OFS="delimiter" in the BEGIN block.

The syntax for reading out the command in a variable output is as follows:

**$ echo | awk '{ "grep root /etc/passwd" | getline cmdout ; print cmdout }'**

awk supports associative arrays, which can use the text as the index.

A for loop is available in awk. It has the following format:

for(i=0;i<10;i++) { print $i ; }

Or:

for(i in array) { print array[i]; }

* length(string): This returns the string length.
* index(string, search\_string): This returns the position at which search\_string is found in the string.
* split(string, array, delimiter): This stores the list of strings generated by using the delimiter in the array.
* substr(string, start-position, end-position): This returns the substring created from the string by using the start and end character offsets.
* sub(regex, replacement\_str, string): This replaces the first occurring regular expression match from the string withreplacment\_str.
* gsub(regex, replacment\_str, string): This is similar to sub(), but it replaces every regular expression match.
* match(regex, string): This returns the result of whether a regular expression (regex) match is found in the string or not. It returns a non-zero output if a match is found, otherwise it returns zero. Two special variables are associated with match(). They are RSTART andRLENGTH. The RSTART variable contains the position at which the regular expression match starts. The RLENGTH variable contains the length of the string matched by the regular expression.

# Finding the frequency of words used in a given file

egrep -o "\b[[:alpha:]]+\b" $filename is used to output only words. The -o option will print the matching character sequence, delimited by a newline character.

# Compressing or decompressing JavaScript

We can use the following command chain to compress a .js file:

**$ cat sample.js | \**

**tr -d '\n\t' | tr -s ' ' \**

**| sed 's:/\\*.\*\\*/::g' \**

**| sed 's/ \?\([{}();,:]\) \?/\1/g'**

We can write a decompression script for making the obfuscated code readable, as follows:

**$ cat obfuscated.txt | sed 's/;/;\n/g; s/{/{\n\n/g; s/}/\n\n}/g'**

Or:

**$ cat obfuscated.txt | sed 's/;/;\n/g' | sed 's/{/{\n\n/g' | sed 's/}/\n\n}/g'**

# Merging multiple files as columns

Usually, the cat command concatenates in a line (or row-wise) fashion.

paste is the command that can be used for column-wise concatenation. The paste command can be used with the following syntax:

**$ paste file1 file2 file3**

The default delimiter is tab. We can also explicitly specify the delimiter by using -d. For example:

**$ paste file1.txt file2.txt -d ","**

# Printing the nth word or column in a file or line

1. To print the fifth column, use the following command:
2. **$ awk '{ print $5 }' filename**
3. We can also print multiple columns and insert our custom string in between columns.

For example, to print the permission and filename of each file in the current directory, use the following set of commands:

**$ ls -l | awk '{ print $1 " : " $8 }'**

**-rw-r--r-- : delimited\_data.txt**

# Printing text between line numbers or patterns

1. To print the lines of a text in a range of line numbers, M to N, use the following syntax:

**$ awk 'NR==M, NR==N' filename.txt**

Or, it can take the stdin input as follows:

**$ cat filename.txt | awk 'NR==6, NR==7'**

To print the lines of a text in a section with start\_pattern and end\_pattern, use the following syntax:

**$ awk '/start\_pattern/, /end \_pattern/' filename**

# Printing lines in the reverse order

tac is the reverse of cat.

The tac syntax is as follows:

**tac file1 file2 …**

1. In tac, \n is the line separator. But, we can also specify our own separator by using the -s "separator" option.
2. We can do it in awk as follows:

**$ seq 9 | \**

**awk '{ lifo[NR]=$0 }**

**END{ for(lno=NR;lno>-1;lno--){ print lifo[lno]; }**

**}'**

\ in the shell script is used to conveniently break a single line command sequence into multiple lines.

# Parsing e-mail addresses and URLs from text

As we are using extended regular expressions (+, for instance), we should use egrep.

**$ egrep -o '[A-Za-z0-9.\_]+@[A-Za-z0-9.]+\.[a-zA-Z]{2,4}' url\_email.txt**

# Removing a sentence in a file containing a word

We will remove the sentence containing the words mobile phones. Use the following sed expression for this task:

**$ sed 's/ [^.]\*mobile phones[^.]\*\.//g' sentence.txt**

# Replacing a pattern with text in all the files in a directory

Let's say we want to replace the text Copyright with the word Copyleft in all .cpp files:

**$ find . -name \*.cpp -print0 | xargs -I{} -0 sed -i 's/Copyright/Copyleft/g' {}**

If you recall, find has an option -exec, which can be used to run a command on each of the files that find will match. We can use this option to achieve the same effect or replace the text with a new one, as follows:

**$ find . -name \*.cpp -exec sed -i 's/Copyright/Copyleft/g' \{\} \;**

Or:

**$ find . -name \*.cpp -exec sed -i 's/Copyright/Copyleft/g' \{\} \+**

While they perform the same function, the first form will call sed once for every file that is found, while in the second form, find will combine multiple filenames and pass them together to sed.

# Text slicing and parameter operations

Replacing some text from a variable can be done as follows:

**$ var="This is a line of text"**

**$ echo ${var/line/REPLACED}**

**This is a REPLACED of text"**

We can produce a substring by specifying the start position and string length, by using the following syntax:

**${variable\_name:start\_position:length}**

To print eight characters starting from the fifth character, use the following command:

**$ echo ${string:4:8}**

The index is specified by counting the start letter as 0. We can also specify counting from the last letter as -1. It is used inside a parenthesis. (-1)is the index for the last letter:

**echo ${string:(-1)}**

**z**

**$ echo ${string:(-2):2}**

**yz**