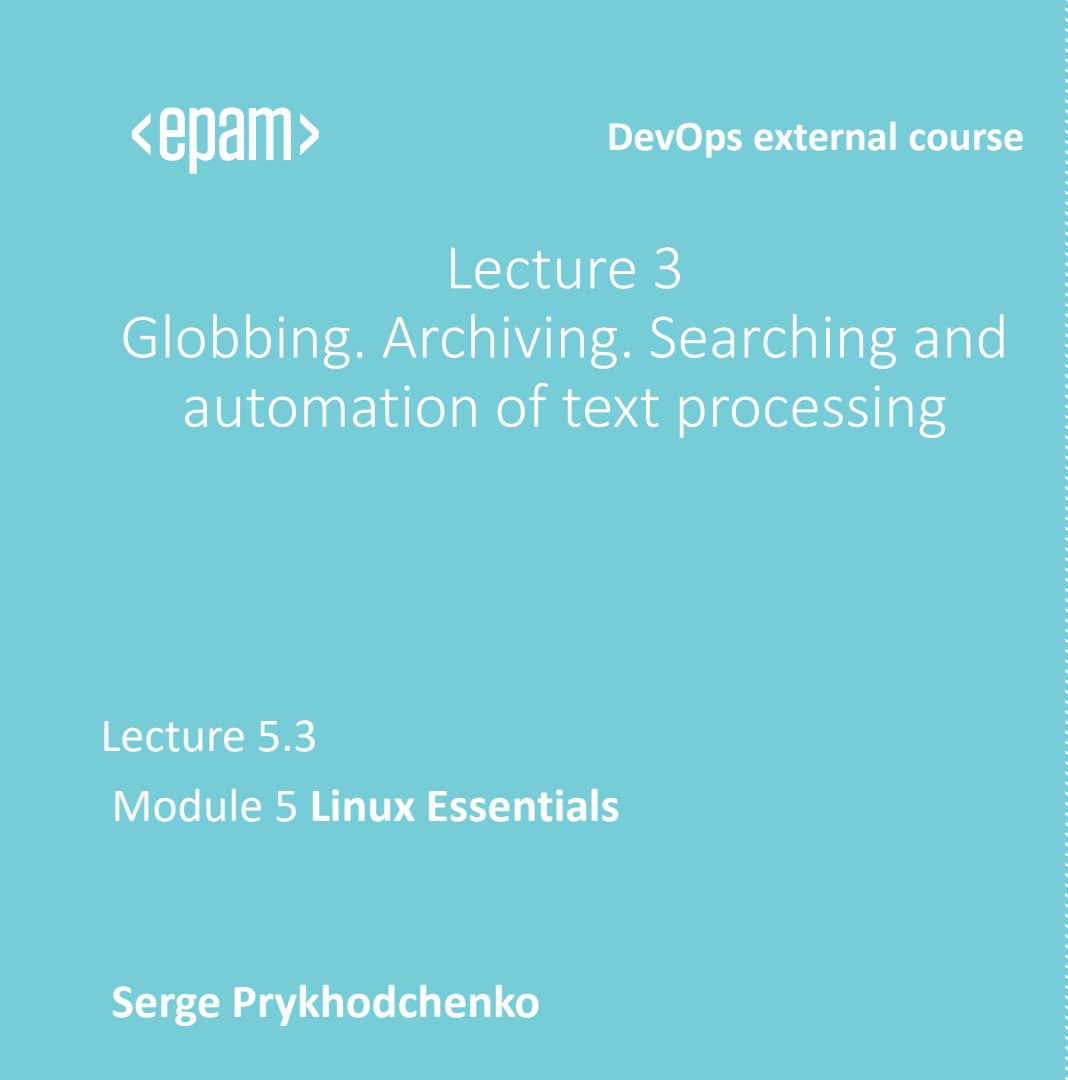


DevOps external course

Lecture 3
Globbing. Archiving. Searching and automation of text processing

Lecture 5.3

Module 5 **Linux Essentials**

Serge Prykhodchenko

Environment Variables and Shell Variables

In Linux and Unix based systems environment variables are a set of dynamic named values, stored within the system that are used by applications launched in shells or subshells. In simple words, an environment variable is a variable with a name and an associated value.

Environment variables allow you to customize how the system works and the behavior of the applications on the system. For example, the environment variable can store information about the default text editor or browser, the path to executable files, or the system locale and keyboard layout settings.

https://linuxize.com/post/how-to-set-and-list-environment-variables-in-linux/

Environment Variables and Shell Variables

Variables have the following format:

KEY=value

KEY="Some other value"

KEY=value1:value2

The names of the variables are case-sensitive. By convention, environment variables should have UPPER CASE names.

When assigning multiple values to the variable they must be separated by the colon : character.

There is no space around the equals = symbol.

Variables can be classified into two main categories, environment variables, and shell variables.

Environment variables are variables that are available system-wide and are inherited by all spawned child processes and shells.

Shell variables are variables that apply only to the current shell instance. Each shell such as zsh and bash, has its own set of internal shell variables.

Environment Variables and Shell Variables

There are several commands available that allow you to list and set environment variables in Linux:

env – The command allows you to run another program in a custom environment without modifying the current one. When used without an argument it will print a list of the current environment variables.

printenv – The command prints all or the specified environment variables.

set – The command sets or unsets shell variables. When used without an argument it will print a list of all variables including environment and shell variables, and shell functions.

unset – The command deletes shell and environment variables.

export – The command sets environment variable

```
MAIL=/var/mail/vagrant
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/us
r/local/games
PWD=/home/vagrant
LANG=en_US.UTF-8
SHLVL=1
XDG_SEAT=seat0
HOME=/home/vagrant
LOGNAME=vagrant
LESSOPEN=| /usr/bin/lesspipe %s
XDG_RUNTIME_DIR=/run/user/1000
LESSCLOSE=/usr/bin/lesspipe %s %s
_=/usr/bin/printenv
vagrant@vagrant-ubuntu-trusty-64:~$ printe_
```

Host-specific system configuration

The /etc hierarchy contains configuration files. A "configuration file" is a local file used to control the operation of a program; it must be static and cannot be an executable binary.
It is recommended that files be stored in subdirectories of /etc rather than directly in /etc.
The following files, or symbolic links to files, must be in /etc if the corresponding subsystem is installed:

Host-specific system configuration

csh.login	Systemwide initialization file for C shell logins (optional)
exports	NFS filesystem access control list (optional)
fstab	Static information about filesystems (optional)
ftpusers	FTP daemon user access control list (optional)
gateways	File which lists gateways for routed (optional)
gettydefs	Speed and terminal settings used by getty (optional)
group	User group file (optional)
host.conf	Resolver configuration file (optional)
hosts	Static information about host names (optional)
hosts.allow	Host access file for TCP wrappers (optional)
hosts.deny	Host access file for TCP wrappers (optional)
hosts.equiv	List of trusted hosts for rlogin, rsh, rcp (optional)
hosts.lpd	List of trusted hosts for lpd (optional)
inetd.conf	Configuration file for inetd (optional)
inittab	Configuration file for init (optional)
issue	Pre-login message and identification file (optional)

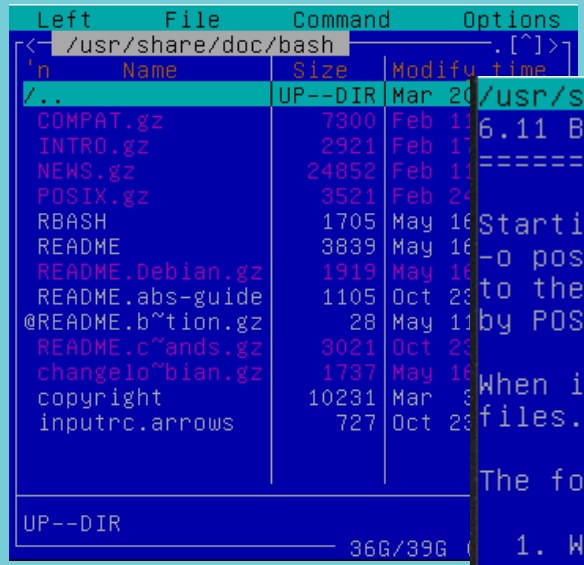
Host-specific system configuration

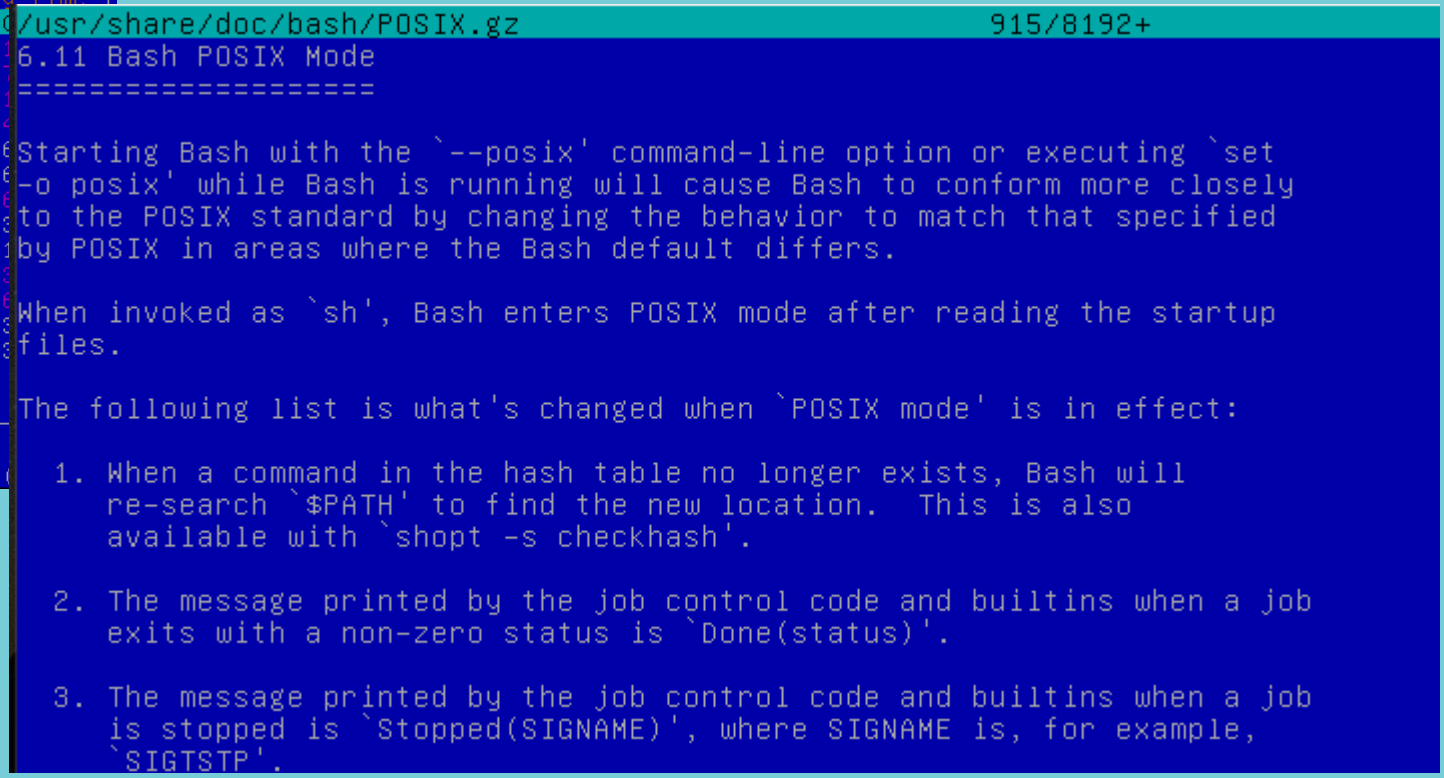
ld.so.conf	List of extra directories to search for shared libraries (optional
motd	Post-login message of the day file (optional)
mtab	Dynamic information about filesystems (optional)
mtools.conf	Configuration file for mtools (optional)
networks	Static information about network names (optional)
passwd	The password file (optional)
printcap	The lpd printer capability database (optional)
profile	Systemwide initialization file for sh shell logins (optional)
protocols	IP protocol listing (optional)
resolv.conf	Resolver configuration file (optional)
rpc	RPC protocol listing (optional)
securetty	TTY access control for root login (optional)
services	Port names for network services (optional)
shells	Pathnames of valid login shells (optional)
syslog.conf	Configuration file for syslogd (optional)

System commands

/usr/bin/lsb_release -ircd # Check the Linux distribution version;
uname -a # Show Linux kernel version;
uname -m # Display computer architecture;
hostname # Show the network name of the computer;
uptime # System uptime without rebooting or shutting down;
shutdown # Shutdown \ reboot:
- ***shutdown -r now*** # reboot;
- ***shutdown -h 20:00*** # Turn off the power at 20:00;
- shutdown -h now # Turn off the power now;
init 0 # Turn off the power;
init 6 # reboot;
halt # Turn off the power;
logout # Log out;
reboot # reboot;

\usr\share\doc





File system usage monitoring

df (disk free)
The df command is the easiest way to get information about the used disk space.

$ df -H

Filesystem	Size	Used	Avail	Use%	Mounted on
dev	2.1G	0	2.1G	0%	/dev
run	2.1G	959k	2.1G	1%	/run
/dev/sda3	64G	24G	38G	38%	/
tmpfs	2.1G	0	2.1G	0%	/dev/shm
tmpfs	2.1G	0	2.1G	0%	/sys/fs/cgroup
tmpfs	2.1G	8.2k	2.1G	1%	/tmp
/dev/sda5	212G	189G	13G	94%	/home
/dev/sda6	160G	38G	115G	25%	/home/data
/dev/sda1	536M	131M	405M	25%	/boot
tmpfs	405M	4.1k	405M	1%	/run/user/1000

File system usage monitoring

du (disk usage)
The du command recursively displays the size of all directories in the specified (current) directory.
The -d parameter specifies the recursion depth.

```
$ du -h -d 1 /home
237M    /home/maxima
138G    /home/oleksii.fedorov
16K     /home/lost+found
14G     /home/doc
8.6G    /home/image
35G     /home/data
2.0G    /home/R
8.3G    /home/install
211G    /home
```

Search by pattern: grep

grep - used to search files for text by pattern:

$ grep template [file ...]

When grep finds a match with a "pattern", it prints the line with the match. Patterns can include regular expressions (egrep).

```
serge@ubserge:~$ grep docker -R
.config/kded_device_automounterrc:LastNameSeen=docker_423.snap
f.txt:Usage:    docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
Binary file nd.tar matches
grep: .bash_history: Permission denied
Binary file .local/share/baloo/index matches
f1.txt:Usage:   docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
grep: .kube: Permission denied
a.txt:Usage:    docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
f2.txt:Usage:   docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Globbing

Bash itself cannot recognize Regular Expressions. Inside scripts, it is commands and utilities -- such as sed and awk -- that interpret RE's.
Bash does carry out filename expansion -- a process known as globbing -- but this does not use the standard RE set. Instead, globbing recognizes and expands wild cards. Globbing interprets the standard wild card characters -- ***** and **?**, character lists in square brackets, and certain other special characters (such as ^ for negating the sense of a match). There are important limitations on wild card characters in globbing, however. Strings containing * will not match filenames that start with a dot, as, for example, .bashrc. Likewise, the ? has a different meaning in globbing than as part of an RE.

ls -l t?.sh
ls -l [ab]*
ls –l [a-c]*
ls -l [^ab]*

Wildcards

- Another function provided by your shell (not your application)
- A quick way to be able to specify multiple related file paths in a single operation
- There are two main wildcards
  - * = Any number of any characters
  - ? = One of any character
- You can include them at any point in a file path and the shell will expand them before passing them on to the program
- Multiple wildcards can be in the same path.
- Command line completion won't work after the first wildcard

Wildcard examples

```
$ ls Monday/*txt
Monday/mon_1.txt  Monday/mon_2.txt  Monday/mon_3.txt  Monday/mon_500.txt

$ ls Monday/mon_?.txt
Monday/mon_1.txt  Monday/mon_2.txt  Monday/mon_3.txt

$ ls */*txt
Friday/fri_1.txt  Monday/mon_1.txt  Monday/mon_3.txt    Tuesday/tue_1.txt
Friday/fri_2.txt  Monday/mon_2.txt  Monday/mon_500.txt  Tuesday/tue_2.txt

$ ls */*1.txt
Friday/fri_1.txt  Monday/mon_1.txt  Tuesday/tue_1.txt
```

Archiving

File archiving and file compression are, by their very nature, different operations. Archiving is the operation of combining multiple files (and directories) into one file in a special format called an archive. Due to the presence of service information in the archive, the size of the archive is larger than the sum of the sizes of the files included in it. Compression is the operation of reducing the size of a file using special algorithms. In UNIX, archiving and compression is usually done by different programs.Tar command is for creating archives in files.
Tar archive flag: file extension ".tar" or letter "t" in extension ".tgz".

Archiving

Archives created by tar command

$tar

Действие	режим	файл
`c` – создать архив	`v` – расширенный режим	`f` – файл
`t` – вывести содержимое архива	`w` – интерактивный режим	
`x` – извлечь	`z` – режим сжатия	
`u` – добавляет только файлы, новее чем те, которые в архиве		
`A` – добавить файлы в существующий архив		
`--delete` – удалить из архива		
`--remove-files` – удалить выходные файлы		
`--exclude=FILE` – исключить файлы из обработки		

Archiving examples

create a.tar archive by placing the a.txt file and the Pictures directory with all its files and subdirectories into it:

```
serge@ubserge:~$ ls
Desktop    Downloads  Pictures  Templates  a.txt  f1.txt  nd.tar
Documents  Music      Public    Videos     f.txt  f2.txt  snap
serge@ubserge:~$ tar -cvf a.tar a.txt Pictures
a.txt
Pictures/
serge@ubserge:~$ ls
Desktop    Downloads  Pictures  Templates  a.tar  f.txt   f2.txt  snap
Documents  Music      Public    Videos     a.txt  f1.txt  nd.tar
serge@ubserge:~$
```

display the contents of the nd.tar archive with an extended presentation of information:

```
serge@ubserge:~$ tar -tvf nd.tar
drwxr-xr-x root/root         0 2020-03-31 20:51 12fd6c3c98e89adcffde66259a8e5e953cf4948a4d2aa537e6ceb0d21ced8a34/
-rw-r--r-- root/root         3 2020-03-31 20:51 12fd6c3c98e89adcffde66259a8e5e953cf4948a4d2aa537e6ceb0d21ced8a34/VERSION
-rw-r--r-- root/root       401 2020-03-31 20:51 12fd6c3c98e89adcffde66259a8e5e953cf4948a4d2aa537e6ceb0d21ced8a34/json
-rw-r--r-- root/root 119202816 2020-03-31 20:51 12fd6c3c98e89adcffde66259a8e5e953cf4948a4d2aa537e6ceb0d21ced8a34/layer.tar
-rw-r--r-- root/root      3427 2020-03-31 20:51 8c8c53c67b75fc265f44a8a9e74b62a06758f663041cbbf905e90dbfa8ef4cd1.json
drwxr-xr-x root/root         0 2020-03-31 20:51 96243fcfd5cf810ae11a00613082b0f6d14271e8d959aef1664c2b8a767140d0/
-rw-r--r-- root/root         3 2020-03-31 20:51 96243fcfd5cf810ae11a00613082b0f6d14271e8d959aef1664c2b8a767140d0/VERSION
-rw-r--r-- root/root       477 2020-03-31 20:51 96243fcfd5cf810ae11a00613082b0f6d14271e8d959aef1664c2b8a767140d0/json
-rw-r--r-- root/root  18655232 2020-03-31 20:51 96243fcfd5cf810ae11a00613082b0f6d14271e8d959aef1664c2b8a767140d0/layer.tar
```

Archiving examples

- extract the prog / scr.sh file from the arh2.tar archive (the prog directory is created if it was not there, and the scr.sh file is written to it from the archive):

$tar –xvf arh2.tar prog/scr.sh

- extract all files from archive arh2.tar :

$tar –xvf arh2.tar

- add echox file to archive:

$tar –uvf arh2.tar echox

Compression

The most widely used file compressor under UNIX is the gzip utility (its reverse utility is gunzip). The attribute of a file compressed with gzip is the extension .gz or .tgz (tar + gzip).

- $ gzip file
- $ gunzip file.gz

When compressing (without additional keys) the original file is deleted, in its place a compressed file is formed, the name of which is equal to the name of the original file plus .gz.

During recovery, the opposite happens: the compressed file is deleted and the initial one is created.The standard compress (uncompress) compressor comes with UNIX.

Working with these utilities is similar to working with gzip (gunzip), but usedextension .Z

For UNIX OS, there are also zip and unzip programs that work similarly to windows versions, including, in addition to compression, archiving and unpacking archives. But, these utilities are not necessarily included in the delivery of the operating system..

Compression

```
serge@ubserge:~$ tar -czf az.tgz a.txt Pictures
serge@ubserge:~$ ls -l
total 135844
drwxr-xr-x 2 serge serge      4096 Apr  9 21:26 Desktop
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Documents
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Downloads
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Music
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Pictures
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Public
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Templates
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Videos
-rw-rw-r-- 1 serge serge     10240 Apr  9 23:38 a.tar
-rw-rw-r-- 3 serge serge      7528 Apr  6 19:24 a.txt
-rw-rw-r-- 1 serge serge      2472 Apr  9 23:53 az.tgz
```

```
serge@ubserge:~$ gzip a.tar
serge@ubserge:~$ ls
Desktop    Downloads  Pictures  Templates  a.tar.gz  az.tgz  f1.txt  nd.tar
Documents  Music      Public    Videos     a.txt     f.txt   f2.txt  snap
serge@ubserge:~$ ls -l
total 135836
drwxr-xr-x 2 serge serge      4096 Apr  9 21:26 Desktop
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Documents
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Downloads
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Music
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Pictures
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Public
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Templates
drwxr-xr-x 2 serge serge      4096 Apr  9 21:24 Videos
-rw-rw-r-- 1 serge serge      2478 Apr  9 23:38 a.tar.gz
-rw-rw-r-- 3 serge serge      7528 Apr  6 19:24 a.txt
-rw-rw-r-- 1 serge serge      2472 Apr  9 23:53 az.tgz
```

semi-colon Operator (;)

The semi-colon operator makes it possible to run, several commands in a single go and the execution of command occurs sequentially.

apt-get update ; apt-get upgrade ; mkdir test

The above command combination will first execute update instruction, then upgrade instruction and finally will create a ‘test‘ directory under the current working directory.

And and Or

AND Operator (&&)
The AND Operator (&&) would execute the second command only, if the execution of first command SUCCEEDS, i.e., the exit status of the first command is 0. This command is very useful in checking the execution status of last command.
For example, I want to visit website tecmint.com using links command, in terminal but before that I need to check if the host is live or not.
ping -c3 www.tecmint.com && links www.tecmint.com

OR Operator (||)
The OR Operator (||) is much like an ‘else‘ statement in programming. The above operator allow you to execute second command only if the execution of first command fails, i.e., the exit status of first command is ‘1‘.
For example, I want to execute ‘apt-get update‘ from non-root account and if the first command fails, then the second ‘links www.tecmint.com‘ command will execute.
apt-get update || links tecmint.com
In the above command, since the user was not allowed to update system, it means that the exit status of first command is ‘1’ and hence the last command ‘links tecmint.com‘ gets executed.
What if the first command is executed successfully, with an exit status ‘0‘? Obviously! Second command won’t execute.
mkdir test || links tecmint.com
Here, the user creates a folder ‘test‘ in his home directory, for which user is permitted. The command executed successfully giving an exit status ‘0‘ and hence the last part of the command is not executed.

PIPE Operator (|)

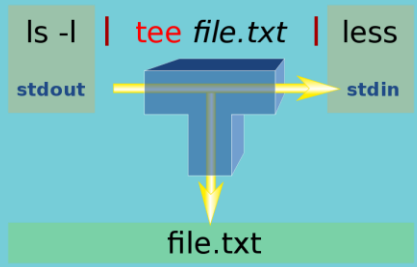
This PIPE operator is very useful where the output of first command acts as an input to the second command. For example, pipeline the output of ‘ls -l‘ to ‘less‘ and see the output of the command.

ls -l | less

echo -e "apple\npear\nbanana"|sort

tee

The tee command reads from the standard input and writes to both standard output and one or more files at the same time. tee is mostly used in combination with other commands through piping.



tee Command Syntax
The syntax for the tee command is as follows:

tee [OPTIONS] [FILE]
Copy
OPTIONS :
-a (--append) - Do not overwrite the files instead append to the given files.
-i (--ignore-interrupts) - Ignore interrupt signals.
Use tee --help to view all available options.
FILE_NAMES - One or more files. Each of which the output data is written to.

wc (Word Count)

The “wc” or the word count command in Bash is considered extremely useful as it helps in finding out various statistics of a file. This command can be used in multiple different variations. However, in this guide, we’re going to learn the basic usage of this command in Bash.

Displaying the Number of Lines, Words, Characters, and the Name of a File:

If you want to display the number of lines, words, characters, and the name of a file in Linux Mint 20, then you can run the “wc” command without any additional flags in this manner:

$ wc File

Printing only the Number of Words and the Name of a File:

If you only want to print the total number of words in a file along with its name, then you can use the “wc” command with the “-w” flag.

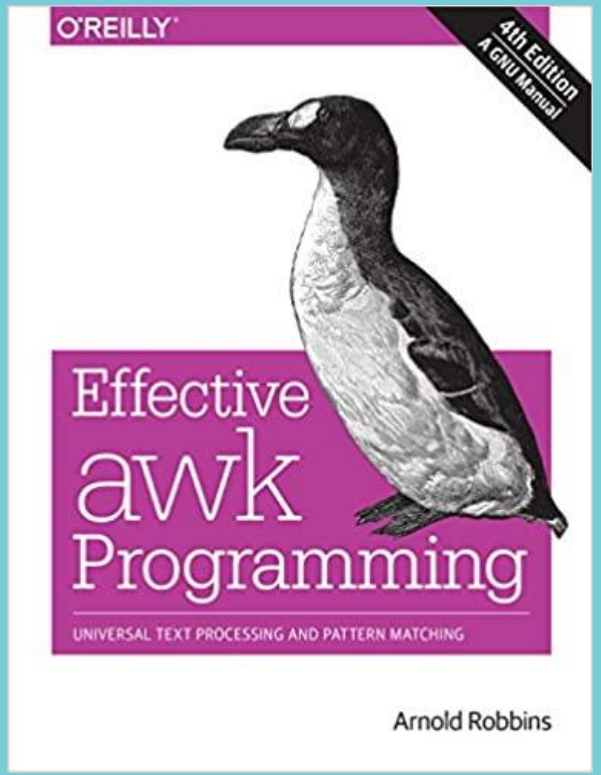
$ wc –w File

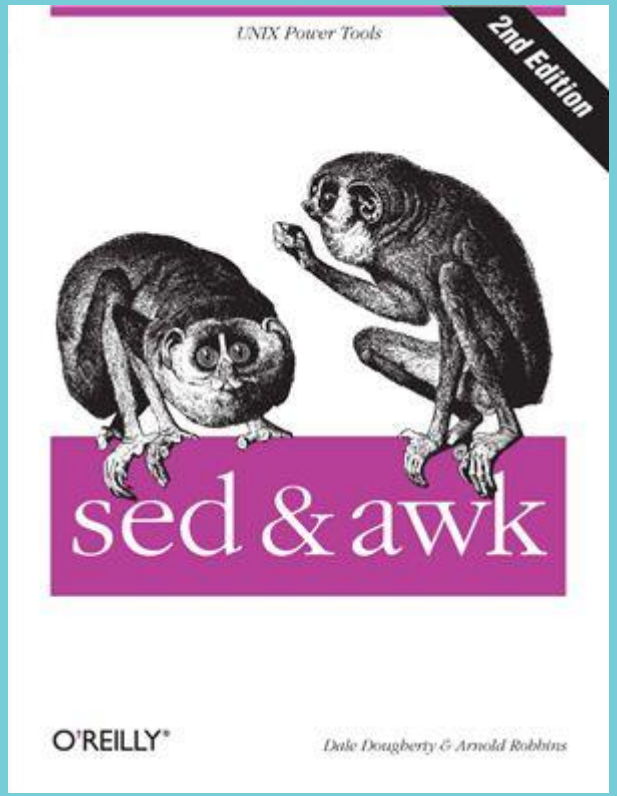
Printing only the Length of the Longest Line (number of characters in the longest line) and the Name of a File:

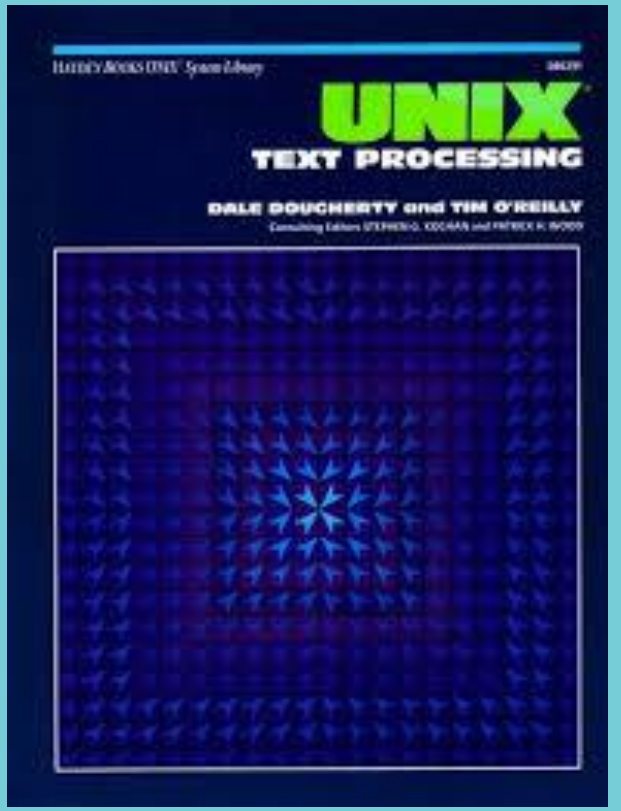
If you just want to display the longest line length and name of a file, then you can use the “wc” command with the “-L” flag.

$ wc –L File

awk


O'REILLY
4th Edition
A GNU Manual
Effective
awk
Programming
UNIVERSAL TEXT PROCESSING AND PATTERN MATCHING
Arnold Robbins


UNIX Power Tools
2nd Edition
sed & awk
O'REILLY
Dale Dougherty & Arnold Robbins


UNIX
TEXT PROCESSING
DALE DOUGHERTY and TIM O'REILLY

awk

awk is a utility / language for extracting data. Awk is often used in conjunction with sed to accomplish a variety of practical text-processing tasks. Like sed, awk reads one line at a time, performs certain actions based on the given options, and prints the result. One of the simplest and most popular uses for awk is to select a column from a text file or from the output of another command. When I installed Debian on my second workstation, I used awk to get a list of those installed on the first machine and feed it to aptitude. I did it using a command like:

```
$ dpkg -l | awk ' {print $2} ' > installed
```

awk: Base

As mentioned, actions performed by awk are enclosed in curly braces, and the entire command is enclosed in single quotes: awk 'condition {action}'. In our example, there are no conditions, but if we want, for example, to select only installed packages related to vim (yes, there is grep, but this is an example, besides, why use two utilities if you can use one), we could dial:

$ dpkg -l | awk ' /'vim'/ {print $2} ‘

This command will list all installed packages containing "vim" in their names. One of the reasons awk is recommended is because it is fast. If I replace "vim" with "lib", I have a list of over 1300 packages on my system.

awk: Base

There are several ways to run awk. If the program is short, then it is easier to run it from the command line.

awk PROGRAM inputfile(s)

If there are many changes, perhaps regularly and for many files, it is easier to put awk commands in a script. The reference to the script is as follows:

awk -f PROGRAM-FILE inputfile(s)

awk: Base

When awk reads a line in a file, it divides the line into fields; this is done using the input field separator specified in the FS variable, which is an awk variable (see the section entitled Output Separators). This variable is predefined and can store one or more space or tab characters.

The variables $ 1, $ 2, $ 3, ..., $ N store the values of the first, second, third, and so on up to the last field of the input string. The variable $ 0 (zero) stores the value of the entire string. This is shown in the figure below, where we see six columns in the data returned by the df command:

```
$ df -l
Filesystem     1K-blocks    Used Available Use% Mounted on
udev              991052       0    991052   0% /dev
tmpfs             204132    1192    202940   1% /run
/dev/sda1       10253588 5063700   4649320  53% /
tmpfs            1020656       0   1020656   0% /dev/shm
tmpfs               5120       4      5116   1% /run/lock
tmpfs            1020656       0   1020656   0% /sys/fs/cgroup
tmpfs             204128       8    204120   1% /run/user/1000
tmpfs             204128       8    204120   1% /run/user/1004
tmpfs             204128      20    204108   1% /run/user/1005
```

awk: Base

```
$ df -l
Filesystem     1K-blocks    Used Available Use% Mounted on
udev             991052       0    991052   0% /dev
tmpfs            204132    1192    202940   1% /run
/dev/sda1      10253588 5063700   4649320  53% /
tmpfs           1020656       0   1020656   0% /dev/shm
tmpfs              5120       4      5116   1% /run/lock
tmpfs           1020656       0   1020656   0% /sys/fs/cgroup
tmpfs            204128       8    204120   1% /run/user/1000
tmpfs            204128       8    204120   1% /run/user/1004
tmpfs            204128      20    204108   1% /run/user/1005
```

The awk print command prints the output from the input file.
The df -l command has nine columns. The print statement uses it like this:

```
[serge@localhost ~]$ df -l|awk '{print $6 "  " $4}'
Использовано%  Использовано
/dev  1995104
/dev/shm  2015316
/run  2014020
/  858624
/tmp  2015232
/boot  756948
/run/user/1000  392604
```

awk: Base

If you use only the output stream separator and do not use formatting, then the output will look rather sad. To make it look significantly better, insert some tabs and extra lines that indicate how to display the data:

```
kelly@octarine ~/test> ls -ldh * | grep -v total | \
awk '{ print "Size is " $5 " bytes for " $9 }'
Size is 160 bytes for orig
Size is 121 bytes for script.sed
Size is 120 bytes for temp_file
Size is 126 bytes for test
Size is 120 bytes for twolines
Size is 441 bytes for txt2html.sh
kelly@octarine ~/test>
```

Note the use of a backslash, which allows input to be wrapped to the next line and the shell does not interpret it as a separate command. Although the length of the command line you enter can be almost unlimited, your monitor screen is limited, and even more so as for the hard copy. Using a backslash will also allow the terminal window to copy and paste lines that are above the current line.

awk: Base

You can print any number of columns and even use them in reverse order. This is demonstrated below for an example of displaying data on the most important sections:

```
kelly@octarine ~> df -h | sort -rnk 5 | head -3 | \
awk '{ print "Partition " $6 "\t: " $5 " full!" }'
Partition /var  : 86% full!
Partition /usr  : 85% full!
Partition /home : 70% full!
```

Subsequence	Meaning
\a	Beep
\n	New line
\t	Tabulation

Print command and regular expressions

Regular expressions can be used as a pattern by enclosing them with slashes. After that, the regular expression will be checked for every entry in the entire text. The syntax is as follows:

awk 'EXPRESSION { PROGRAM }' file(s)

awk: Base

The following example displays information about local drives only, network filesystems are not shown:

kelly is in ~> df -h | awk '/dev\/hd/ { print $6 "\t: " $5 }'

```
/       : 46%
/boot   : 10%
/opt    : 84%
/usr    : 97%
/var    : 73%
/.vol1  : 8%
```

Here's another example where we use regular expressions to search the /etc directory for files ending with ".conf" and starting with either "a" or "x":

kelly is in /etc> ls -l | awk '/\<(a|x).*\.conf$/ { print $9 }'

```
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
```

This example illustrates a special property of the dot character used in regular expressions: the first dot indicates that we want to find any character after the first search string, for the second dot its special property is disabled because it is part of the search string (end of file name).

awk: Examples

Syntax	Description
awk ' {print $1,$3} '	Print only the first and third columns using stdin
awk ' {print $0} '	Print all columns using stdin
awk ' /'pattern'/ {print $2} '	Print only elements of the second column that match "pattern" using stdin
awk -f script.awk inputfile	Like sed, awk uses the -f switch to get instructions from a file, which is useful when there are a lot of them and it is impractical to enter them manually in a terminal.
awk ' program ' inputfile	Executes program using data from inputfile
awk "BEGIN { print \"Hello, world!!\" }"	Classical "Hello, world" by awk
awk '{ print }'	Prints whatever is entered from the command line until EOF is encountered
#! /bin/awk -f BEGIN { print "Hello, world!" }	Awk script for the classic "Hello, world!" (make it executable with chmod and run)

awk: Examples

# This is a program that prints \ "Hello, world!" # and exits	Comments in awk scripts
awk -F "" 'program' files	Defines the field separator to be null, as opposed to the default white space
awk -F "regex" 'program' files	The field separator can also be a regular expression
awk '{ if (length($0) > max) max = \ length($0) } END { print max }' inputfile	Prints the length of the longest line
awk 'length($0) > 80' inputfile	Print all lines longer than 80 characters
awk 'NF > 0' data	Prints every line containing at least one field (NF stands for Number of Fields)
awk 'BEGIN { for (i = 1; i <= 7; i++) print int(101 * rand()) }'	Prints seven random numbers ranging from 0 to 100
ls -l . | awk '{ x += $5 } ; END \ { print "total bytes: " x }' total bytes: 7449362	Prints the total number of bytes used by files in the current directory
ls -l . | awk '{ x += $5 } ; END \ { print "total kilobytes: " (x + \ 1023)/1024 }' total kilobytes: 7275.85	Prints the total number of kilobytes used by files in the current directory

awk: Examples

awk -F: '{ print $1 }' /etc/passwd | sort	Prints a sorted list of usernames
awk 'END { print NR }' inputfile	Prints the number of lines in the file, NR stands for Number of Rows
awk 'NR % 2 == 0' data	Prints even lines of a file.
ls -l | awk '$6 == "Nov" { sum += $5 } END { print sum }'	Prints the total number of bytes of the file that was last edited in November.
awk '$1 ~/J/' inputfile	Regular expression for all entries in the first field that start with a capital j.
awk '$1 ~!/J/' inputfile	Regular expression for all entries in the first field that do not start with a capital j.
awk 'BEGIN { print "He said \"hi!\" \to her." }'	Escaping double quotes in awk.
echo aaaabcd | awk '{ sub(/a+/, \"<A>"); print }'	Print "<A>bcd"
awk '{ $2 = $2 - 10; print $0 }' inventory	Modifies inventory and prints it with the difference that the value of the second field will be decreased by 10.
awk '{ $6 = ($5 + $4 + $3 + $2); print \ $6' inventory	Even if field six does not exist in inventory, you can create it and assign a value, then output it.

awk: Examples

echo a b c d | awk '{ OFS = ":"; $2 = "" > print $0; print NF }'	OFS is the Output Field Separator and the command will print "a :: c: d" and "4" because although the second field is canceled, it still exists, so it can be counted.
echo a b c d | awk '{ OFS = ":"; \ $2 = ""; $6 = "new" > print $0; print NF }'	Another example of creating a field; as you can see, the field between $ 4 (existing) and $ 6 (created) will also be created (as empty $ 5), so the output will look like "a :: c: d :: new" "6".
echo a b c d e f | awk '\ { print "NF =", NF; > NF = 3; print $0 }'	Dropping three fields (last) by changing the number of fields.
FS=[ ]	This is a regex to set space as field separator.
echo ' a b c d ' | awk 'BEGIN { FS = \ "[\t\n]+" } > { print $2 }'	Only prints "a".
awk -n '/RE/{p;q;}' file.txt	Print only the first match of the regular expression.
awk -F\\\\ '...' inputfiles ...	Sets the field separator to \\
BEGIN { RS = "" ; FS = "\n" } { print "Name is:", $1 print "Address is:", $2 print "City and State are:", $3 print "" }	If we have an entry like "John Doe 1234 Unknown Ave. Doeville, MA", this script sets the field separator to a newline so that it can easily work with strings.

awk: Examples

awk 'BEGIN { OFS = ";"; ORS = "\n\n" } > { print $1, $2 }' inputfile	If the file contains two fields, the records will be printed as: "field1:field2 field3;field4 ...;..." since the output field separator is two new lines, and the field separator is ";".
awk 'BEGIN { > OFMT = "%.0f" # print numbers as \ integers (rounds) > print 17.23, 17.54 }'	17 and 18 will be printed because the Output ForMaT is rounding floating point numbers to the nearest integer value.
awk 'BEGIN { > msg = "Dont Panic!" > printf "%s\n", msg >} '	You can use printf in much the same way as in C.
awk '{ printf "%-10s %s\n", $1, \ $2 }' inputfile	Prints the first field as a 10-character, left-justified string, followed by the second field as normal.
awk '{ print $2 > "phone-list" }' \inputfile	A simple example of data extraction where the second field is written under the name "phone-list".
awk '{ print $1 > "names.unsorted" command = "sort -r > names.sorted" print $1 | command }' inputfile	Writes the names contained in $ 1 to a file, then sort and output the result to another file.
awk 'BEGIN { printf "%d, %d, %d\n", 011, 11, \ 0x11 }' Will print 9, 11, 17 if (/foo/ || /bar/) print "Found!"	Simple search for foo or bar.
awk '{ sum = $2 + $3 + $4 ; avg = sum / 3 > print $1, avg }' grades	Simple arithmetic operations (most like C)

awk: Examples

awk '{ print "The square root of", \ $1, "is", sqrt($1) }' 2 The square root of 2 is 1.41421 7 The square root of 7 is 2.64575	Simple extensible calculator
awk '$1 == "start", $1 == "stop"' inputfile	Prints each entry between start and stop.
awk ' > BEGIN { print "Analysis of \"foo\"" } > /foo/ { ++n } > END { print "\"foo\" appears", n,\ "times." }' inputfile	The BEGIN and END rules are executed only once, before and after each processing of the record.
echo -n "Enter search pattern: " read pattern awk "/$pattern/ "'{ nmatches++ } END { print nmatches, "found" }' inputfile Search using shell if (x % 2 == 0) print "x is even" else print "x is odd"	Simple condition. awk, like C, also supports the?: operators.
awk '{ i = 1 while (i <= 3) { print $i i++ } }' inputfile	Prints the first three fields of each record, one per line.
awk '{ for (i = 1; i <= 3; i++) print $i }'	Prints the first three fields of each record, one per line.

awk: Examples

BEGIN { if (("date" | getline date_now) <= 0) { print "Can't get system date" > \ "/dev/stderr" exit 1 } print "current date is", date_now close("date") }	Exiting with an error code other than 0 means something is wrong.
awk 'BEGIN { > for (i = 0; i < ARGC; i++) > print ARGV[i] > }' file1 file2	Prints awk file1 file2
for (i in frequencies) delete frequencies[i]	Removes elements in an array
foo[4] = "" if (4 in foo) print "This is printed, even though foo[4] \ is empty"	Check array elements
function ctime(ts, format) { format = "%a %b %d %H:%M:%S %Z %Y" if (ts == 0) ts = systime() # use current time as default return strftime(format, ts) }	The awk version of the ctime () function in C. This is how you can define your own functions in awk.
BEGIN { _cliff_seed = 0.1 } function cliff_rand() { _cliff_seed = (100 * log(_cliff_seed)) % 1 if (_cliff_seed < 0) _cliff_seed = - _cliff_seed return _cliff_seed }	Cliff random number generator.

Sed

Sed is a non-interactive line editor. It takes text from either stdin or a text file, performs some string operations, and then prints the result to stdout or a file. Typically in scripts, sed is used in pipelining, along with other commands and utilities.

Sed determines, based on a given address space, on which lines to perform operations. The address space of lines is specified either by their ordinal numbers or by a pattern. For example, 3d command will cause sed to delete the third line, while / windows / d means that all lines containing windows should be deleted.

Of the whole variety of operations, we will focus on the three most commonly used. These are p for printing (to stdout), d for deleting, and s for replacing.

Sed

`[диапазон строк]/p`	print	Печать указанного `[диапазона строк]`
`[диапазон строк]/d`	delete	Удалить указанный `[диапазон строк]`
`s/pattern1/pattern2/`	substitute	Заменить первое встреченное соответствие шаблону `pattern1`, в строке, на `pattern2`
`[диапазон строк]/s/pattern1/pattern2/`	substitute	Заменить первое встреченное соответствие шаблону `pattern1`, на `pattern2`, в указанном диапазоне строк
`[диапазон строк]/y/pattern1/pattern2/`	transform	заменить любые символы из шаблона `pattern1` на соответствующие символы из `pattern2`, в указанном диапазоне строк (эквивалент команды `tr`)
`g`	global	Операция выполняется над всеми найденными соответствиями внутри каждой из заданных строк

Sed

Without the g (global) operator, the replacement operation will be performed only for the first match found with the given pattern on each line.
In some cases, sed operations must be enclosed in quotation marks.
sed -e '/^$/d' $filename
The -e switch indicates that a line follows that should be interpreted as a set of editing instructions. Strong quotation marks (‘…’) prevent the shell from interpreting regexp characters as special characters. Actions are performed on the lines contained in the file $filename.
In some cases, editing commands do not work in single quotation marks.
filename=file1.txt
pattern=BEGIN
sed "/^$pattern/d" "$filename" # will delete all lines with BEGIN

sed '/^$pattern/d' "$filename"
gives a different result. In this case, in strong quotes ('...'), the value of the $ pattern variable is not substituted.

Sed

Sed uses the -e switch to determine that the next line is an editing instruction, or set of instructions.
If the statement is the only one, then the use of this key is optional.

sed -n '/xzy/p' $filename

The -n switch forces sed to print only those lines that match the specified pattern.
Otherwise (without the -n switch), all lines will be displayed.
Here, the -e switch is optional, since there is only one command

Sed

`8d`	Удалить 8-ю строку.
`/^$/d`	Удалить все пустые строки.
`11,$ d`	Показать первые 10 строк.
`1,/^$/d`	Удалить все строки до первой пустой строки, включительно.
`/Jones/p`	Вывести строки, содержащие "`Jones`" (с ключом `-n`).
`s/Windows/Linux/`	В каждой строке, заменить первое встретившееся слово `Windows` на слово `Linux`.
`s/BSOD/stability/g`	В каждой строке, заменить все встретившиеся слова `BSOD` на `stability`.
`s/ *$//`	Удалить все пробелы в конце каждой строки.
`s/00*/0/g`	Заменить все последовательности ведущих нулей одним символом `0`.
`/GUI/d`	Удалить все строки, содержащие `GUI`.
`s/GUI//g`	Удалить все найденные `GUI`, оставляя остальную часть строки без изменений.

Regular Expressions

An expression is a string of characters. Those characters having an interpretation above and beyond their literal meaning are called *metacharacters*. A quote symbol, for example, may denote speech by a person, *ditto*, or a meta-meaning for the symbols that follow. Regular Expressions are sets of characters and/or metacharacters that match (or specify) patterns.

A Regular Expression contains one or more of the following:

•*A character set*. These are the characters retaining their literal meaning. The simplest type of Regular Expression consists *only* of a character set, with no metacharacters.

•*An anchor*. These designate (*anchor*) the position in the line of text that the RE is to match. For example, ^, and $ are anchors.

•*Modifiers*. These expand or narrow (*modify*) the range of text the RE is to match. Modifiers include the asterisk, brackets, and the backslash.

Regular Expressions

The main uses for Regular Expressions (REs) are text searches and string manipulation. An RE matches a single character or a set of characters -- a string or a part of a string.

The asterisk -- * -- matches any number of repeats of the character string or RE preceding it, including zero instances.

"1133*" matches 11 + one or more 3's: 113, 1133, 1133333, and so forth.

The dot -- . -- matches any one character, except a newline. [2]

"13." matches 13 + at least one of any character (including a space): 1133, 11333, but not 13 (additional character missing).

The caret -- ^ -- matches the beginning of a line, but sometimes, depending on context, negates the meaning of a set of characters in an RE.

The dollar sign -- $ -- at the end of an RE matches the end of a line.

"XXX$" matches XXX at the end of a line.

"^$" matches blank lines.

Regular Expressions

The main uses for Regular Expressions (REs) are text searches and string manipulation. An RE matches a single character or a set of characters -- a string or a part of a string.

Brackets -- [...] -- enclose a set of characters to match in a single RE.
"[xyz]" matches any one of the characters x, y, or z.
"[c-n]" matches any one of the characters in the range c to n.
"[B-Pk-y]" matches any one of the characters in the ranges B to P and k to y.
"[a-z0-9]" matches any single lowercase letter or any digit.
"[^b-d]" matches any character except those in the range b to d. This is an instance of ^ negating or inverting the meaning of the following RE (taking on a role similar to ! in a different context).
Combined sequences of bracketed characters match common word patterns. "[Yy][Ee][Ss]" matches yes, Yes, YES, yEs, and so forth. "[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" matches any Social Security number.

Regular Expressions

The main uses for Regular Expressions (REs) are text searches and string manipulation. An RE matches a single character or a set of characters -- a string or a part of a string.

The backslash -- \ -- escapes a special character, which means that character gets interpreted literally (and is therefore no longer special).
A "\$" reverts back to its literal meaning of "$", rather than its RE meaning of end-of-line.
Likewise a "\\" has the literal meaning of "\".
Escaped "angle brackets" -- \<...\> -- mark word boundaries.
The angle brackets must be escaped, since otherwise they have only their literal character meaning.
"\<the\>" matches the word "the," but not the words "them," "there," "other," etc.

https://tldp.org/LDP/abs/html/x17129.html

QUESTIONS & ANSWERS


THANK YOU!