1. **Editors history**
2. **Vi, vim and Nano**
3. **Create file, edit content and save**

**20:00**

**Text editors AND Text Manipulation**

**Irrespective of the role you play with Linux (system administrator, developer or user), you often need to browse through and parse text files, and/or extract data from them.**

**These are file manipulation operations. Thus, it is essential for the Linux user to become adept at performing certain operations on files.**

**Most of the time, such file manipulation is done at the command line, which allows users to perform tasks more efficiently than while using a GUI. Furthermore, the command line is more suitable for automating often executed tasks.**

**Indeed, experienced system administrators write customized scripts to accomplish such repetitive tasks, standardized for each particular environment. We will discuss such scripting later in much detail.**

**In this section, we will concentrate on command line file and text manipulation-related utilities.**

**At some point, you will need to manually edit text files. You might be composing an email off-line, writing a script to be used for bash or other command interpreters, altering a system or application configuration file, or developing source code for a programming language such as C, Python or Java.**

**Linux administrators may sidestep using a text editor, instead employing graphical utilities for creating and modifying system configuration files. However, this can be more laborious than directly using a text editor, and be more limited in capability. Note that word processing applications (including those that are part of common office application suites) are not really basic text editors; they add a lot of extra (usually invisible) formatting information that will probably render system administration configuration files unusable for their intended purpose. So, knowing how to confidently use one or more text editors is really an essential skill to have for Linux.**

**By now, you have certainly realized Linux is packed with choices; when it comes to text editors, there are many choices, ranging from quite simple to very complex, including:**

**nano**

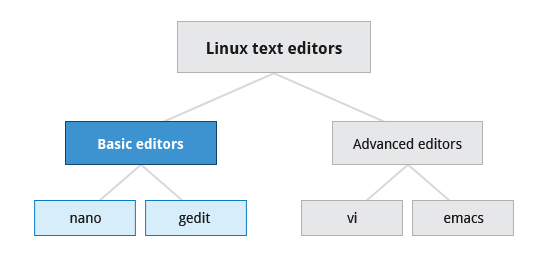
**gedit**

**vi**

**emacs.**

**Vim**

**In this section, we learn first about the nano and gedit editors, which are relatively simple and easy to learn, and then later the more complicated choices, vi and emacs. Before we start, let us take a look at some cases where an editor is not needed.**



**vi and emacs**

**Developers and administrators experienced in working on UNIX-like systems almost always use one of the two venerable editing options: vi and emacs. Both are present or easily available on all distributions and are completely compatible with the versions available on other operating systems.**

**Both vi and emacs have a basic purely text-based form that can run in a non-graphical environment. They also have one or more graphical interface forms with extended capabilities; these may be friendlier for a less experienced user. While vi and emacs can have significantly steep learning curves for new users, they are extremely efficient when one has learned how to use them.**

**You need to be aware that fights among seasoned users over which editor is better can be quite intense and are often described as a holy war.**

**vi and emacs**

**Usually, the actual program installed on your system is vim, which stands for Vi IMproved, and is aliased to the name vi. The name is pronounced as “vee-eye”.**

**Even if you do not want to use vi, it is good to gain some familiarity with it: it is a standard tool installed on virtually all Linux distributions. Indeed, there may be times where there is no other editor available on the system.**

**GNOME extends vi with a very graphical interface known as gvim and KDE offers kvim. Either of these may be easier to use at first.**

**When using vi, all commands are entered through the keyboard. You do not need to keep moving your hands to use a pointer device such as a mouse or touchpad, unless you want to do so when using one of the graphical versions of the editor.**

**vimtutor**

Typing **vimtutor** launches a short but very comprehensive tutorial for those who want to learn their first **vi** commands. Even though it provides only an introduction and just seven lessons, it has enough material to make you a very proficient **vi** user, because it covers a large number of commands. After learning these basic ones, you can look up new tricks to incorporate into your list of vi commands because there are always more optimal ways to do things in **vi** with less typing.

**Modes in vi**

vi provides three modes, as described in the table below. It is vital to not lose track of which mode you are in. Many keystrokes and commands behave quite differently in different modes.

**Command**

* **By default, vi starts in Command mode.**
* **Each key is an editor command.**
* **Keyboard strokes are interpreted as commands that can modify file contents.**

**Insert**

* **Type i to switch to Insert mode from Command mode.**
* **Insert mode is used to enter (insert) text into a file.**
* **Insert mode is indicated by an “? INSERT?” indicator at the bottom of the screen.**
* **Press Esc to exit Insert mode and return to Command mode.**

**Line**

* **Type: to switch to the Line mode from Command mode. Each key is an external command, including operations such as writing the file contents to disk or exiting.**
* **Uses line editing commands inherited from older line editors. Most of these commands are actually no longer used. Some line editing commands are very powerful.**
* **Press Esc to exit Line mode and return to Command mode.**

**Working with Files in vi**

**The table describes the most important commands used to start, exit, read, and write files in vi. The ENTER key needs to be pressed after all of these commands.**

**vi myfile =>** Start the editor and edit **myfile**

**vi -r myfile=>** Start and edit **myfile** in recovery mode from a system crash

**:r file2 =>** Read in **file2** and insert at current position

**:w =>** Write to the file

**:w myfile =>** Read in **file2** and insert at current position

**:w! file2 =>** Overwrite **file2**

**:x or :wq =>** Exit and write out modified file

**:q =>** Quit

**:q! =>** Quit even though modifications have not been saved

**Changing Cursor Positions in vi**

**The table describes the most important keystrokes used when changing cursor position in**vi**. Line mode commands (those following colon:) require the**ENTER**key to be pressed after the command is typed.**

**arrow keys =>** To move up, down, left and right

**j or <ret> =>** To move one line down

**k =>** To move one line up

**h** or Backspace => To move one character left

**l** or Space => To move one character right

**0 =>** To move to beginning of line

**$ =>** To move to end of line

**W =>** To move to beginning of next word

**:0** or **1G =>** To move to beginning of file

**:n** or **nG =>** To move to line n

**:$** or **G =>** To move to last line in file

**CTRL-F** or **Page Down =>** To move forward one page

**CTRL-B** or **Page Up =>** To move backward one page

**^l =>** To refresh and center screen

**Searching for Text in vi**

The table describes the most important commands used when searching for text in vi. The **ENTER** key should be pressed after typing the search pattern.

**/pattern** => Search forward for pattern

**?pattern** => Search backward for pattern

The table describes the most important *keystrokes* used when searching for text in vi.

**n =>** Move to next occurrence of search pattern

**N =>** Move to previous occurrence of search pattern

**Working with Text in vi**

**the table describes the most important keystrokes used when changing, adding, and deleting text in vi.**

**a =>** Append text after cursor; stop upon **Escape** key

**A =>** Append text at end of current line; stop upon **Escape** key

**I =>** Insert text before cursor; stop upon **Escape** key

**I =>** Insert text at beginning of current line; stop upon **Escape** key

**o =>** Start a new line below current line, insert text there; stop upon **Escape** key

**O =>** Start a new line above current line, insert text there; stop upon **Escape** key

**r =>** Replace character at current position

**R =>** Replace text starting with current position; stop upon **Escape** key

**x =>** Delete character at current position

**Nx =>** Delete N characters, starting at current position

**dw =>** Delete the word at the current position

**D =>** Delete the rest of the current line

**dd =>** Delete the current line

**Ndd or dNd =>** Delete N lines

**u =>** Undo the previous operation

**yy =>** Yank (copy) the current line and put it in buffer

**Nyy or yNy =>** Yank (copy) N lines and put it in buffer

**P =>** Paste at the current position the yanked line or lines from the buffer

**Vi Editor Short cuts:**

**Esc i => for insert mode**

**Esc shift colon wq! => for save and quit form the editor.**

**Esc shift colon x! => for save and quit from the editor,**

**Esc shift zz => for save and quit from the editor.**

**Esc shift colon q! => quit without save if the file is modified or uninterrupted.**

**Esc a => to add the text after the cursor  
esc shift a => to add the text at the end of the line**

**Esc r => to replace the character**

**Esc shift r => to replace the entire line**

**Esc o => to add the line below the cursor**

**Esc shift o => to add the line above the cursor.**

**Esc cw => to change the word**

**Esc dw => to delete the word**

**Esc x => to delete the character**

**Esc dd => to delete the line**

**Esc ndd => to delete the no of lines**

**Esc u => to undo changes**

**Esc yy => to copy the line**

**Esc p => to paste**

**Esc nyy => to copy the no of lines**

**Esc p => to paste**

**Esc shift colon set nu => to set the line number in the file**

**Esc shift colon set nonu => to remove the line numbers**

**Esc shift colon line number => to move cursor to the particular line number.**

**Esc shift colon /<word> => to move the cursor to the specified word.**

**Esc shift colon 1, $ s/<old word> / <new word >/ g => to replace the word in the entire file.**

**Shift + ctrl ++ => to increase the font**

**Shift + ctrl - - => to decrease the font**

**Alt + f10 => full screen**

**Ctrl +l clear screen**

**Commands for vi**

Commands Used to Start, Exit, Read and Write Files in vi

**Command Usage**

vi myfile Start the vi editor and edit the myfile file

vi -r myfile Start vi and edit myfile in recovery mode from a system crash

:r file2 <ret> Read in file2 and insert at current position

:w<ret> Write to the file

:w myfile Write out the file to myfile

:w! file2<ret> Overwrite file2

:x or :wq Exit vi and write out modified file

:q Quit vi

:q! Quit vi even though modifications have not been saved

Keystrokes Used When Changing Cursor Position in vi

**Working with Large Files**

**System administrators need to work with configuration files, text files, documentation files, and log files. Some of these files may be large or become quite large as they accumulate data with time. These files will require both viewing and administrative updating. In this section, you will learn how to manage such large files.**

**For example, a banking system might maintain one simple large log file to record details of all of one day's ATM transaction. Due to a security attack or a malfunction, the administrator might be forced to check for some data by navigating within the file. In such cases, directly opening the file in an editor will cause issues, due to high memory utilization, as an editor will usually try to read the whole file into memory first. However, one can use less to view** the **contents of such a large file, scrolling up and down page by page, without the system having to place the** **entire file in memory before starting. This is much faster than using a text editor**

**Viewing somefile can be done by typing either of the two following commands:**

$ less somefile  
$ cat somefile | less

**By default, man pages are sent through the less command. You may have encountered the older more utility which has the same basic function but fewer capabilities: i.e. less is more!**

head

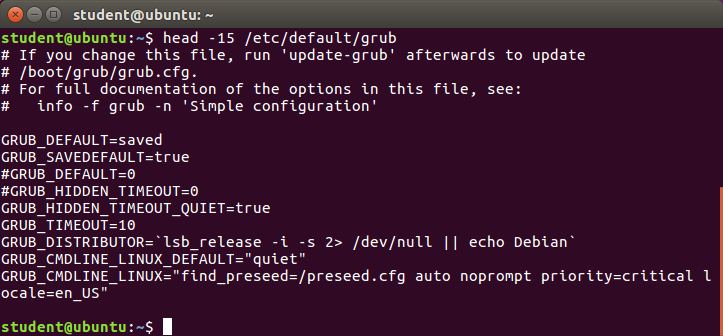
**head reads the first few lines of each named file (10 by default) and displays it on standard output. You can give a different number of lines in an option.**

**For example, if you want to print the first 5 lines from /etc/default/grub, use the following command:**

$ head –n 5 /etc/default/grub

You can also just say:

head -5 /etc/default/grub



**tail**

**tail prints the last few lines of each named file and displays it on standard output. By default, it displays the last 10 lines. You can give a different number of lines as an option. tail is especially useful when you are troubleshooting any issue using log files, as you probably want to see the most recent lines of output.**

**For example, to display the last 15 lines of somefile.log, use the following command:**

**$ tail -n 15 somefile.log**

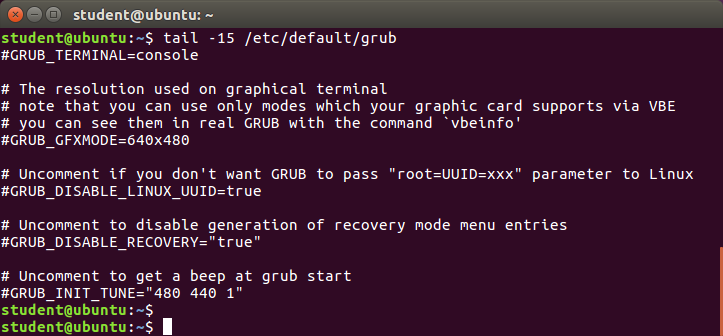
**You can also just say:**

**tail -15 somefile.log**

**To continually monitor new output in a growing log file:**

**$ tail -f somefile.log**

**This command will continuously display any new lines of output in somefile.log as soon as they appear. Thus, it enables you to monitor any current activity that is being reported and recorded.**



**Introduction to sed and awk**

**It is very common to create and then repeatedly edit and/or extract contents from a file. Let’s learn how to use sed and awk to easily perform such operations.**

**Note that many Linux users and administrators will write scripts using comprehensive scripting languages such as**Python**and**perl**, rather than use**sed**and**awk**(and some other utilities we will discuss later). Using such utilities is certainly fine in most circumstances; one should always feel free to use the tools one is experienced with. However, the utilities that are described here are much lighter; i.e. they use fewer system resources, and execute faster. There are situations (such as during booting the system) where a lot of time would be wasted using the more complicated tools, and the system may not even be able to run them. So, the simpler tools will always be needed**.

sed

sed**is a powerful text processing tool and is one of the oldest, earliest and most popular UNIX utilities. It is used to modify the contents of a file or input stream, usually placing the contents into a new file or output stream. Its name is an abbreviation for**s**tream**ed**itor.**

sed**can filter text, as well as perform substitutions in data streams.**

**Data from an input source/file (or stream) is taken and moved to a working space. The entire list of operations/modifications is applied over the data in the working space and the final contents are moved to the standard output space (or stream).**

sed Command Syntax

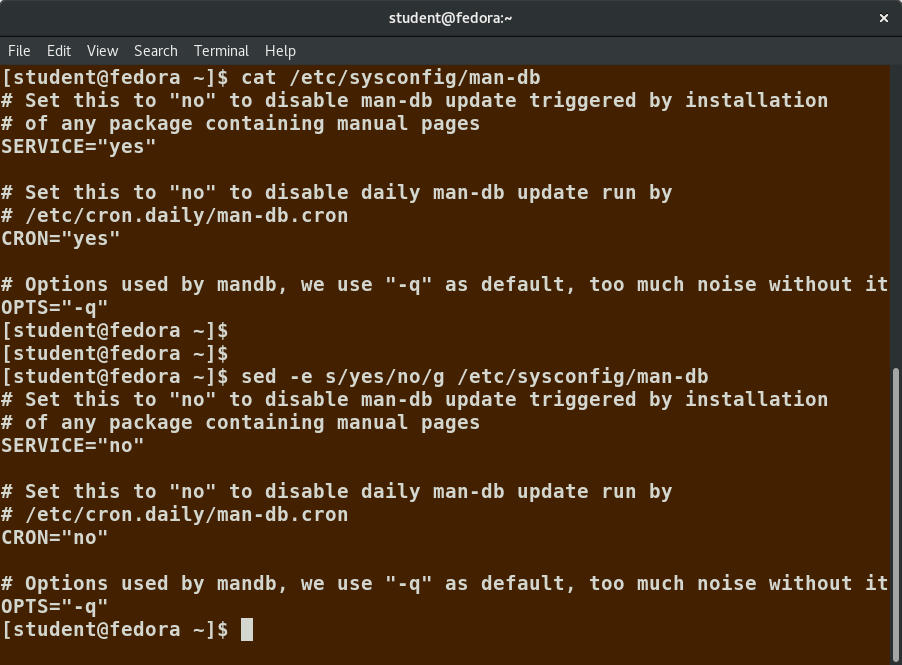
**You can invoke sed using commands like those listed in the accompanying table.**

**sed -e command <filename> => Specify editing commands at the command line, operate on file and put the output on standard out (e.g. the terminal)**

**sed -f scriptfile <filename> =>** Specify a scriptfile containing **sed** commands, operate on file and put output on standard out

**echo "I hate you" | sed s/hate/love/ =>** Use **sed** to filter standard input, putting output on standard out

The **-e** option allows you to specify multiple editing commands simultaneously at the command line. It is unnecessary if you only have one operation invoked.



sed Basic Operations

Now that you know that you can perform multiple editing and filtering operations with **sed**, let’s explain some of them in more detail. The table explains some basic operations, where **pattern** is the current string and **replace\_string** is the new string:

**sed s/pattern/replace\_string/ file =>** Substitute first string occurrence in every line

**sed s/pattern/replace\_string/g file =>** Substitute all string occurrences in every line

**sed 1,3s/pattern/replace\_string/g file =>** Substitute all string occurrences in a range of lines

**sed -i s/pattern/replace\_string/g file =>** Save changes for string substitution in the same file

You must use the **-i** option with care, because the action is not reversible. It is always safer to use **sed** without the **–i** option and then replace the file yourself, as shown in the following example:

**$ sed s/pattern/replace\_string/g file1 > file2**

The above command will replace all occurrences of **pattern** with **replace\_string** in **file1** and move the contents to **file2**. The contents of **file2** can be viewed with **cat file2**. If you approve, you can then overwrite the original file with **mv file2 file1**.

Example: To convert **01/02/…** to **JAN/FEB/…**

**sed -e 's/01/JAN/' -e 's/02/FEB/' -e 's/03/MAR/' -e 's/04/APR/' -e 's/05/MAY/' \**  
**-e 's/06/JUN/' -e 's/07/JUL/' -e 's/08/AUG/' -e 's/09/SEP/' -e 's/10/OCT/' \**  
**-e 's/11/NOV/' -e 's/12/DEC/'**

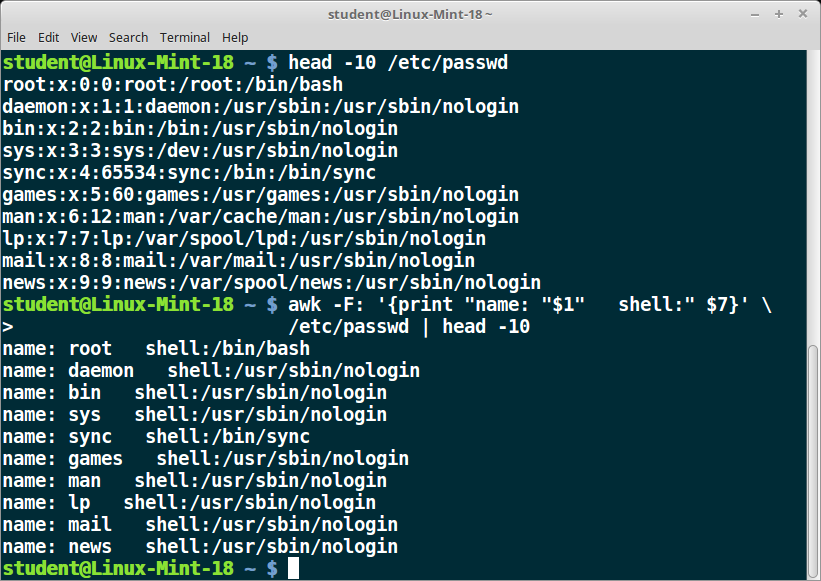
awk

**awk** is used to extract and then print specific contents of a file and is often used to construct reports. It was created at Bell Labs in the 1970s and derived its name from the last names of its authors: Alfred **A**ho, Peter **W**einberger, and Brian**K**ernighan.

**awk** has the following features:

* + 1. It is a powerful utility and interpreted programming language.
    2. It is used to manipulate data files, and for retrieving and processing text.
    3. It works well with fields (containing a single piece of data, essentially a column) and records (a collection of fields, essentially a line in a file).

**awk** is invoked as shown in the following:



As with **sed**, short **awk** commands can be specified directly at the command line, but a more complex script can be saved in a file that you can specify using the **-f** option.

**awk ‘command’  file =>** Specify a command directly at the command line

**awk -f scriptfile file =>** Specify a file that contains the script to be executed

**awk Basic Operations**

**The table explains the basic tasks that can be performed using awk. The input file is read one line at a time, and, for each line, awk matches the given pattern in the given order and performs the requested action. The -F option allows you to specify a particular field separator character. For example, the /etc/passwd file uses ":" to separate the fields, so the -F: option is used with the /etc/passwd file.**

**The command/action in awk needs to be surrounded with apostrophes (or single-quote (')). awk can be used as follows:**

**awk '{ print $0 }' /etc/passwd** => Print entire file

**awk -F: '{ print $1 }' /etc/passwd =>** Print first field (column) of every line, separated by a space

**awk -F: '{ print $1 $7 }' /etc/passwd =>** Print first and seventh field of every line

Lab 13.1: Using sed

**Search for all instances of the user command interpreter (shell) equal to /sbin/nologin in /etc/passwd and replace them with /bin/bash.**

**Click the link below to view a solution to the Lab exercise.**

File Manipulation Utilities

In managing your files, you may need to perform tasks such as sorting data and copying data from one location to another. Linux provides numerous file manipulation utilities that you can use while working with text files. In this section, you will learn about the following file manipulation programs:

* + - **sort**
    - **uniq**
    - **paste**
    - **join**
    - **split**.

You will also learn about regular expressions and search patterns.

sort

**sort**is used to rearrange the lines of a text file, in either ascending or descending order according to a sort key. You can also sort with respect to particular fields (columns) in a file. The default sort key is the order of the ASCII characters (i.e. essentially alphabetically).

**sort** can be used as follows:

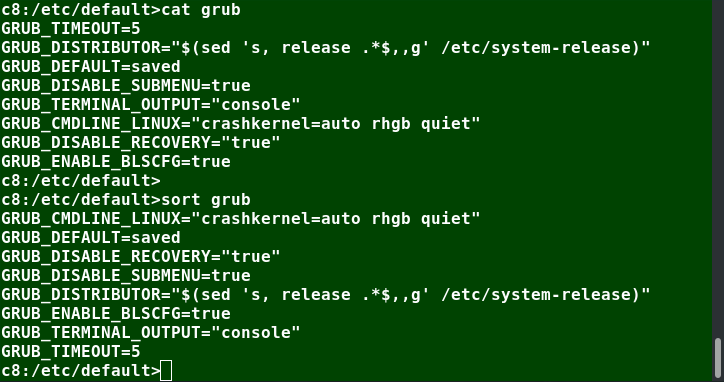
**sort <filename> =>** Sort the lines in the specified file, according to the characters at the beginning of each line

**cat file1 file2 | sort =>** Combine the two files, then sort the lines and display the output on the terminal

**sort -r <filename> =>** Sort the lines in reverse order

**sort -k 3 <filename> =>** Sort the lines by the 3rd field on each line instead of the beginning

When used with the **-u** option, **sort** checks for unique values after sorting the records (lines). It is equivalent to running **uniq** (which we shall discuss) on the output of sort.



**sort**

uniq

**uniq** removes duplicate consecutive lines in a text file and is useful for simplifying the text display.

Because **uniq** requires that the duplicate entries must be consecutive, one often runs sort first and then pipes the output into **uniq**; if sort is used with the **-u** option, it can do all this in one step.

To remove duplicate entries from multiple files at once, use the following command:

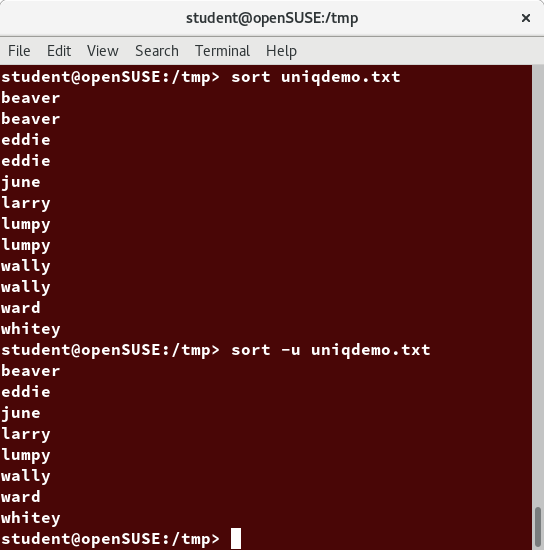
**sort file1 file2 | uniq > file3**

or

**sort -u file1 file2 > file3**

To count the number of duplicate entries, use the following command:

**uniq -c filename**



**uniq**

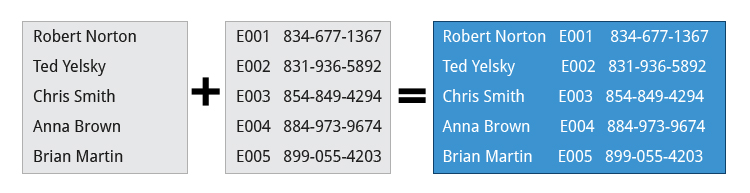
paste

Suppose you have a file that contains the full name of all employees and another file that lists their phone numbers and Employee IDs. You want to create a new file that contains all the data listed in three columns: name, employee ID, and phone number. How can you do this effectively without investing too much time?

**paste** can be used to create a single file containing all three columns. The different columns are identified based on delimiters (spacing used to separate two fields). For example, delimiters can be a blank space, a tab, or an **Enter**. In the image provided, a single space is used as the delimiter in all files.

**paste** accepts the following options:

* + - **-d** delimiters, which specify a list of delimiters to be used instead of tabs for separating consecutive values on a single line. Each delimiter is used in turn; when the list has been exhausted, **paste** begins again at the first delimiter.
    - **-s**, which causes paste to append the data in series rather than in parallel; that is, in a horizontal rather than vertical fashion.



**paste**

Using paste

**paste** can be used to combine fields (such as name or phone number) from different files, as well as combine lines from multiple files. For example, line one from **file1** can be combined with line one of **file2**, line two from **file1** can be combined with line two of **file2**, and so on.

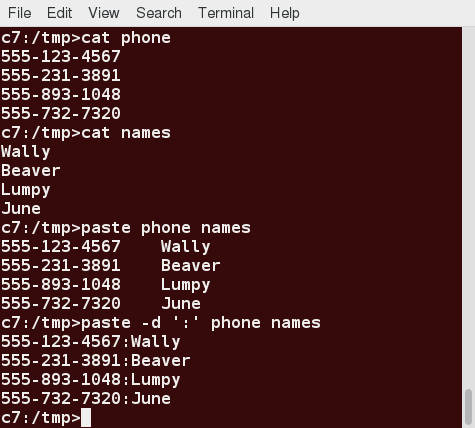
To paste contents from two files one can do:

**$ paste file1 file2**

The syntax to use a different delimiter is as follows:

**$ paste -d, file1 file2**

Common delimiters are 'space', 'tab', '|', 'comma', etc.

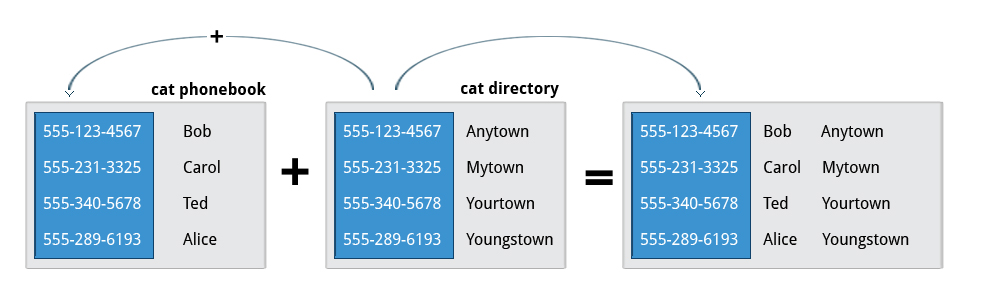


**Using paste**

join

Suppose you have two files with some similar columns. You have saved employees’ phone numbers in two files, one with their first name and the other with their last name. You want to combine the files without repeating the data of common columns. How do you achieve this?

The above task can be achieved using **join**, which is essentially an enhanced version of **paste**. It first checks whether the files share common fields, such as names or phone numbers, and then joins the lines in two files based on a common field.

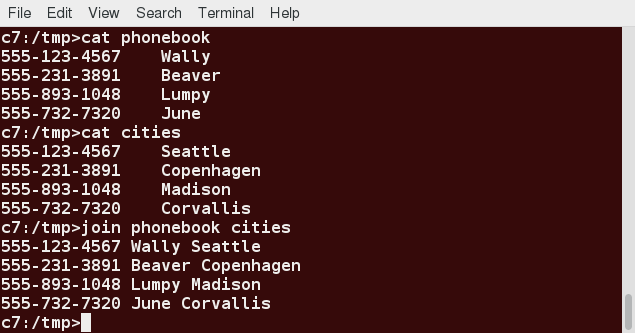


**join**

Using join

To combine two files on a common field, at the command prompt type **join file1 file2** and press the **Enter** key.

For example, the common field (i.e. it contains the same values) among the **phonebook** and **cities** files is the phone number, and the result of joining these two files is shown in the screen capture.

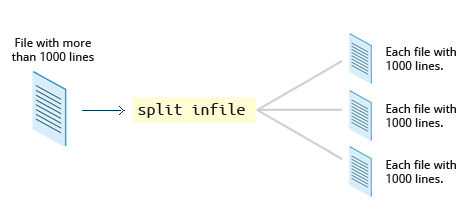


**Using join**

split

**split** is used to break up (or split) a file into equal-sized segments for easier viewing and manipulation, and is generally used only on relatively large files. By default, **split** breaks up a file into 1000-line segments. The original file remains unchanged, and a set of new files with the same name plus an added prefix is created. By default, the **x** prefix is added. To split a file into segments, use the command **split infile**.

To split a file into segments using a different prefix, use the command **split infile <Prefix>**.



**split**

Using split

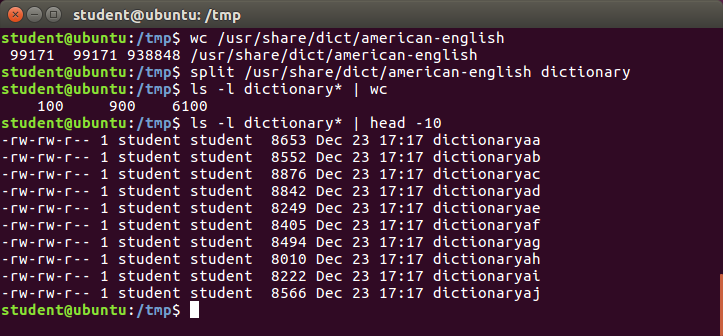
We will apply **split** to an American-English dictionary file of over 99,000 lines:

**$ wc -l american-english  
99171 american-english**

where we have used **wc** (word count, soon to be discussed) to report on the number of lines in the file. Then, typing:

**$ split american-english dictionary**

will split the American-English file into 100 equal-sized segments named **dictionary*xx***. The last one will of course be somewhat smaller.



**Using split**

Regular Expressions and Search Patterns

**Regular expressions** are text strings used for matching a specific pattern, or to search for a specific location, such as the start or end of a line or a word. Regular expressions can contain both normal characters or so-called meta-characters, such as **\*** and **$**.

Many text editors and utilities such as **vi**, **sed**, **awk**, **find** and **grep** work extensively with regular expressions. Some of the popular computer languages that use regular expressions include Perl, Python and Ruby. It can get rather complicated and there are whole books written about regular expressions; thus, we will do no more than skim the surface here.

These regular expressions are different from the wildcards (or meta-characters) used in filename matching in command shells such as bash

(Which were covered in *Chapter 7: Command-Line Operations*).

The table lists search patterns and their usage.

**.(dot) =>** Match any single character

**a|z =>** Match a or z

**$ =>** Match end of a line

**^ =>** Match beginning of a line

* => Match preceding item 0 or more times

Using Regular Expressions and Search Patterns

For example, consider the following sentence: **the quick brown fox jumped over the lazy dog**.

Some of the patterns that can be applied to this sentence are as follows:

**a.. =>** matches azy

**b.|j. =>** matches both br and ju

**..$ =>** matches og

**l.\* =>** matches lazy dog

**l.\*y =>** matches lazy

**the.\* =>** matches the whole sentence

Lab 13.2: Parsing Files with awk (and sort and uniq)

Generate a column containing a unique list of all the shells used for users in **/etc/passwd**.

You may need to consult the manual page for **/etc/passwd** as in:

**student:/tmp> man 5 passwd**

Which field in **/etc/passwd** holds the account’s default shell (user command interpreter)?

How do you make a list of unique entries (with no repeats)?

Click the link below to view a solution to the Lab exercise.

**Solution**

Parsing Files with awk (and sort and uniq)

The field in **/etc/passwd** that holds the shell is number 7. To display the field holding the shell in **/etc/passwd** using **awk** and produce a unique list:

**$ awk -F: '{print $7}' /etc/passwd | sort -u**

or

**$ awk -F: '{print $7}' /etc/passwd | sort | uniq**

For example:

**$ awk -F: '{print $7}' /etc/passwd | sort -u**

**/bin/bash**

**/bin/sync**

**/sbin/halt**

**/sbin/nologin**

**/sbin/shutdown**

grep

**grep**is extensively used as a primary text searching tool. It scans files for specified patterns and can be used with regular expressions, as well as simple strings, as shown in the table:

**grep [pattern] <filename> =>** Search for a pattern in a file and print all matching lines

**grep -v [pattern] <filename> =>** Print all lines that do**not** match the pattern

**grep [0-9] <filename> =>** Print the lines that contain the numbers **0** through **9**

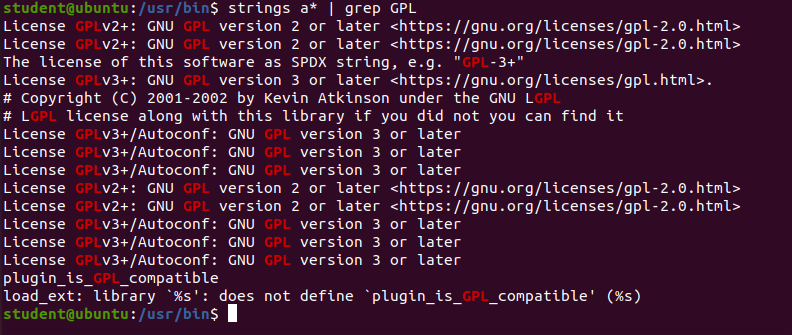
**grep -C 3 [pattern] <filename> =>** Print context of lines (specified number of lines above and below the pattern) for matching the pattern. Here, the number of lines is specified as 3

 strings

**strings**are used to extract all printable character strings found in the file or files given as arguments. It is useful in locating human-readable content embedded in binary files; for text files one can just use **grep**.

For example, to search for the string **my\_string**in a spreadsheet:  
  
**$ strings book1.xls | grep my\_string**

The screenshot shows a search of a number of programs to see which ones have GPL licenses of various versions.



**strings**

Lab 13.3: Using grep

In the following we give some examples of things you can do with the **grep** command; your task is to experiment with these examples and extend them.

* 1. Search for your username in file **/etc/passwd** .
  2. Find all entries in **/etc/services** that include the string **ftp**:
  3. Restrict to those that use the tcp protocol.
  4. Now restrict to those that do not use the tcp protocol, while printing out the line number
  5. Get all strings that start with **ts** or end with **st**.

Click the link below to view a solution to the Lab exercise.

**Solution:**

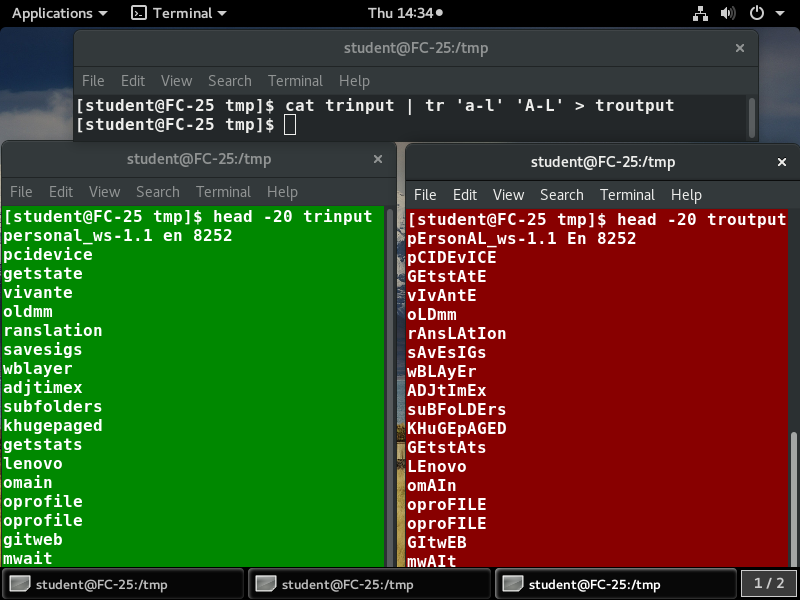
Lab Solution: Using grep

1. **student:/tmp> grep student /etc/passwd**
2. **student:/tmp> grep ftp /etc/services**
3. **student:/tmp> grep ftp /etc/services | grep tcp**
4. **student:/tmp> grep -n ftp /etc/services | grep -v tcp**
5. **student:/tmp> grep -e ^ts -e st$ /etc/services**

[Miscellaneous Text Utilities](https://learning.edx.org/course/course-v1:LinuxFoundationX+LFS101x+2T2021/block-v1:LinuxFoundationX+LFS101x+2T2021+type@sequential+block@a2314bbb95e54581a49f835fb64f618b)

tr

In this section, you will learn about some additional text utilities that you can use for performing various actions on your Linux files, such as changing the case of letters or determining the count of words, lines, and characters in a file.



The **tr** utility is used to translate specified characters into other characters or to delete them. The general syntax is as follows:

**$ tr [options] set1 [set2]**

The items in the square brackets are optional. **tr** requires at least one argument and accepts a maximum of two. The first, designated **set1** in the example, lists the characters in the text to be replaced or removed. The second, **set2**, lists the characters that are to be substituted for the characters listed in the first argument. Sometimes these sets need to be surrounded by apostrophes (or single-quotes (')) in order to have the shell ignore that they mean something special to the shell. It is usually safe (and may be required) to use the single-quotes around each of the sets as you will see in the examples below.

For example, suppose you have a file named **city** containing several lines of text in mixed case. To translate all lower-case characters to upper case, at the command prompt type **cat city | tr a-z A-Z** and press the **Enter** key.

**tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ =>** Convert lower case to upper case

**tr '{}' '()' < inputfile > outputfile =>** Translate braces into parenthesis

**echo "This is for testing" | tr [:space:] '\t' =>** Translate white-space to tabs

**echo "This   is   for    testing" | tr -s [:space:] =>** Squeeze repetition of characters using **-s**

**echo "the geek stuff" | tr -d 't' =>** Delete specified characters using **-d** option

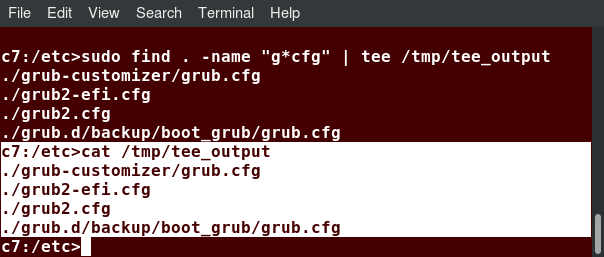
**echo "my username is 432234" | tr -cd [:digit:] =>** Complement the sets using **-c** option

tee

**tee** takes the output from any command, and, while sending it to standard output, it also saves it to a file. In other words, it ***tees*** the output stream from the command: one stream is displayed on the standard output and the other is saved to a file.

For example, to list the contents of a directory on the screen and save the output to a file, at the command prompt type **ls -l | tee newfile** and press the **Enter** key.

Typing **cat newfile** will then display the output of **ls –l**.



wc

**wc** (**w**ord **c**ount) counts the number of lines, words, and characters in a file or list of files. Options are given in the table below.

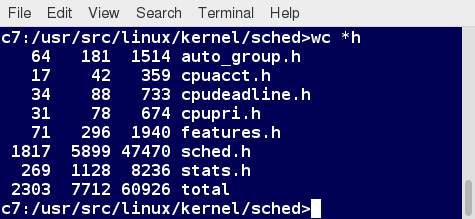
**–l =>** Displays the number of lines

**-c =>** Displays the number of bytes

**-w =>** Displays the number of words

By default, all three of these options are active.

For example, to print only the number of lines contained in a file, type **wc -l filename** and press the **Enter** key.



**wc**

cut

**cut** is used for manipulating column-based files and is designed to extract specific columns. The default column separator is the **tab** character. A different delimiter can be given as a command option.

For example, to display the third column delimited by a blank space, at the command prompt type **ls -l | cut -d" " -f3** and press the **Enter** key.



Lab 13.4: Using tee

The **tee** utility is very useful for saving a copy of your output while you are watching it being generated.

Execute a command such as doing a directory listing of the **/etc** directory:

**student:/tmp> ls -l /etc**

while both saving the output in a file and displaying it at your terminal.

Click the link below to view a solution to the Lab exercise.

Lab Solution: Using tee

**student:/tmp> ls -l /etc | tee /tmp/ls-output**

**student:/tmp> less /tmp/ls-output**

**total 2948**

**drwxr-xr-x. 3 root root 4096 Nov 3 07:27 abrt**

**-rw-r--r--. 1 root root 16 Jan 15 2015 adjtime**

**-rw-r--r-- 1 root root 1518 Jun 7 2013 aliases**

**-rw-r--r--. 1 root root 12288 Nov 3 07:49 aliases.db**

**drwxr-xr-x. 2 root root 4096 Nov 3 07:26 alsa**

**drwxr-xr-x. 2 root root 4096 Jan 20 07:28 alternatives**

**-rw------- 1 root root 541 Feb 23 2016 anacrontab**

**-rw-r--r-- 1 root root 55 Jun 6 2016 asound.conf**

**-rw-r--r-- 1 root root 1 May 23 2016 at.deny**

**drwxr-xr-x. 2 root root 4096 Nov 3 07:26 at-spi2**

**drwxr-x---. 3 root root 4096 Nov 3 07:26 audisp**

**drwxr-x---. 3 root root 4096 Nov 3 07:26 audit**

**drwxr-xr-x. 4 root root 4096 Nov 3 07:32 avahi**

**drwxr-xr-x. 2 root root 4096 Jan 18 06:59 bash\_completion.d**

**-rw-r--r-- 1 root root 2853 May 4 2016 bashrc**

**drwxr-xr-x. 2 root root 4096 Nov 7 10:20 binfmt.d**

**drwxr-xr-x 2 root root 4096 Nov 3 07:26 bluetooth**

**drwxr-xr-x. 2 root root 4096 Apr 9 2015 bonobo-activation**

**drwxr-xr-x 2 root root 12288 Nov 3 07:26 brltty**

**-rw-r--r-- 1 root root 21929 May 6 2016 brltty.conf**

**-rw-r--r-- 1 root root 676 Jun 23 2016 cgconfig.conf**

**:**

Lab 13.5: Using wc

Using **wc** (word count), find out how many lines, words, and characters there are in all the files in **/var/log** that have the **.log** extension.

Click the link below to view a solution to the Lab exercise.

Lab Solution: Using wc

**student:/tmp> wc /var/log/\*.log**

**325 2204 18114 /var/log/boot.log**

**0 0 0 /var/log/pm-powersave.log**

**3098 17871 125630 /var/log/systemtap.log**

**5234 251967 4889056 /var/log/vbox-install.log**

**wc: /var/log/wpa\_supplicant.log: Permission denied**

**1313 15374 99968 /var/log/Xorg.0.log**

**150 1185 8520 /var/log/Xorg.1.log**

**318 2851 20868 /var/log/Xorg.9.log**

**wc: /var/log/yum.log: Permission denied**

**10438 291452 5162156 total**

You have completed Chapter 13. Let’s summarize the key concepts covered:

* + - The command line often allows the users to perform tasks more efficiently than the GUI.
    - **cat**, short for concatenate, is used to read, print, and combine files.
    - **echo**displays a line of text either on standard output or to place in a file.
    - **sed**is a popular stream editor often used to filter and perform substitutions on files and text data streams.
    - **awk**is an interpreted programming language, typically used as a data extraction and reporting tool.
    - **sort**is used to sort text files and output streams in either ascending or descending order.
    - **uniq** eliminates duplicate entries in a text file.
    - **paste**combines fields from different files. It can also extract and combine lines from multiple sources.
    - **join** combines lines from two files based on a common field. It works only if files share a common field.
    - **split**breaks up a large file into equal-sized segments.
    - Regular expressions are text strings used for pattern matching. The pattern can be used to search for a specific location, such as the start or end of a line or a word.
    - **grep** searches text files and data streams for patterns and can be used with regular expressions.
    - **tr** translates characters, copies standard input to standard output, and handles special characters.
    - **tee** saves a copy of standard output to a file while still displaying at the terminal.
    - **wc** (word count) displays the number of lines, words, and characters in a file or group of files.
    - **cut** extracts columns from a file.
    - **less**views files a page at a time and allows scrolling in both directions.
    - **head** displays the first few lines of a file or data stream on standard output. By default, it displays 10 lines.
    - **tail** displays the last few lines of a file or data stream on standard output. By default, it displays 10 lines.
    - **strings** extracts printable character strings from binary files.
    - The **z** command family is used to read and work with compressed files.

**dealing with editor**

$ sudo vim nano wget -y

$ vim file1

i = insertion mode

readonly files

$ vim /etc/ssh/sshd\_config => no permission to view

$ vim /etc/passwd => readonly file, only you can view the file.

:q! => forcible quit

**vim shortcuts**

**gg** => takes you to first line in file

**shift+G** => move the cursor to bottom

**dd** => deletes the line

**uu** => undo

**ctrl+r** => redo

**ddd**=> deletes 3 lines down to cursor

**copy and paste lines**

**yy** => copy

**p** => paste the content below the cursor

**P** => paste the content above the cursor

**yyyy** =>4 lines copied

**search word**

linux is case sensitive

/XXXX => search a word in a file

/20 => 20 in a file will be highlighted

:2 => go to line 2 form anywhere.

:5 => go to line 5 form anywhere.

$ cat file1

$ cat -n file1

w => word by word you can go on text .

**o** => down the cursor one down to line

**O** => top the cursor one up in the line

a => Starting of the line

A => End of the line

$ vi test.sh => .sh extension is optional => it simply represents to identify the shell script.

$ bash test.sh

=> through script we can do lot of automation. we can reuse the script.

**How to Copy, Cut and Paste in Vim / Vi**

When working with text files, copying, cutting, and pasting text is one of the most commonly performed tasks.

**Vim** or its precursor Vi comes preinstalled on macOS and almost all Linux distributions. Knowing the basics of Vim is helpful in a situation where your favourite editor is not available.

This article shows how to copy, cut, and paste in Vim / Vi editor.

When you launch the Vim editor, you’re in the normal mode. In this mode, you can run Vim commands and navigate through the file.

To go back to normal mode from any other mode, just press the Esc key.

Vim has its own terminology for copying, cutting, and pasting. Copy is called yank (y), cut is called delete (d), and paste is called put (p).

**Copying (Yanking)**

To copy text, place the cursor in the desired location and press the y key followed by the movement command. Below are some helpful yanking commands:

yy - Yank (copy) the current line, including the newline character.

3yy - Yank (copy) three lines, starting from the line where the cursor is positioned.

y$ - Yank (copy) everything from the cursor to the end of the line.

y^ - Yank (copy) everything from the cursor to the start of the line.

Yw  - Yank (copy) to the start of the next word.

yiw  – Yank (copy) the current word.

y% - Yank (copy) to the matching character. By default, supported pairs are (), {}, and []. Useful to copy text between matching brackets.

**Cutting (Deleting)**

In normal mode, d is the key for cutting (deleting) text. Move the cursor to the desired position and press the d key, followed by the movement command. Here are some helpful deleting commands:

dd - Delete (cut) the current line, including the newline character.

3dd - Delete (cut) three lines, starting from the line where the cursor is positioned,

d$ - Delete (cut) everything from the cursor to the end of the line.

The movement commands that apply for yanking are also valid for deleting. For example, dw, deletes to the start of the next word, and d^ deletes everything from the cursor to the start of the line.

**Pasting (Putting)**

To put the yanked or deleted text, move the cursor to the desired location and press p to put (paste) the text after the cursor or P to put (paste) before the cursor.

**Copy, Cut, and Paste in Visual Mode**

Vim’s visual mode allows you to select and manipulate text.

Place the cursor on the line you want to begin copping or cutting.

The visual mode has three subtypes.

Press v to enter the visual mode.

Press V to enter visual line mode, where the text is selected by line.

Press Ctrl+v to enter visual block mode. In this mode, the text is selected by rectangle blocks.

Entering the visual mode also marks a starting selection point.

Move the cursor to the end of the text you want to copy or cut. You can use a movement command or up, down, right, and left arrow keys.

Press y to copy, or d to cut the selection.

Move the cursor to the location where you want to paste the contents.

Press P to paste the contents before the cursor, or p to paste it after the cursor.

In this guide, we have shown you how to copy, cut, and paste in Vim.

How to Show Line Numbers in Vim / Vi

**Vim/Vi is the text editor of choice for many software developers and Linux system administrators.**

**By default, Vim doesn’t show line numbers, but they can be easily turned on. Vim supports three modes of line numbering that helps you navigate through the files. In addition to the standard absolute line numbering, Vim also supports relative and hybrid line numbering modes.**

**This article will show you how to display or hide line numbers in Vim / Vi text editor.**

**Besides helping navigate the code, line numbering is also useful in other situations such as pair programming, debugging scripts, code reviews, referring to a specific line, and more.**

Absolute Line Numbers

The absolute line numbering is the standard line numbering, which displays the appropriate line number next to each line of text.

1. Press the Esc key to switch to command mode.
2. Press : (colon) and the cursor will move at the bottom left corner of the screen. Type set number or set nu and hit Enter.

:set number

Line numbers will be displayed at the left side of the screen:

To disable the absolute line numbers off, run the :set nonumber or set nonu commands:

:set nonumber

Copy

You can also toggle the line numbers with :set number! or :set nu!:

:set number!

Copy

Relative Line Numbers

When the relative line numbering is enabled, the current line is shown as 0; The lines above and below from the current line are incrementally numbered (1, 2, 3, etc.).

Relative line mode is handy because many Vim operations, such as moving up/down and deleting lines work on relative line numbers.

For example, to delete the next ten lines below the cursor, you would use the d10j command. With relative line numbers enabled you’ll have a better visual overview on the code.

To enable the relative line numbering, switch to the command mode and enter :set relativenumber or :set rnu:

:set relativenumber

To disable the relative line numbering, type :set norelativenumber or set nornu:

:set nonumber

To toggle the relative line numbering, use the :set relativenumber! or :set rnu! command:

:set number!

Hybrid Line Numbers

In Vim 7.4 and later, enabling both the absolute and relative line numbers at the same time sets up the hybrid line number mode.

Hybrid line numbering is the same as the relative line numbering with the only difference being that the current line instead of showing 0 shows its absolute line number.

To turn on the hybrid line numbering, run both the number and relativenumber commands:

:set number relativenumber

The same can be achieved by running the commands one by one:

:set number

:set relativenumber

To disable the hybrid mode, you’ll need to turn off both the absolute and relative numbering.

Permanent Settings

If you want line numbers to appear each time you launch Vim, add the appropriate command to your .vimrc (Vim configuration file). For example, to enable absolute line numbering, you would add the following:

vim ~/.vimrc

:set number

**Conclusion**

**To show line numbers in Vim, use the: set number command for absolute line numbers, set relative number for relative line numbers. If both absolute and relative line numbers are enabled Vim switches to the hybrid line numbering mode.**

**8. Standard file streams**

**stdin -> 0 -> keyboard**

**stdout -> 1 -> terminal**

**stderr -> 2 -> logfile**

**9. I/O Redirection**

**$ do\_something > inputfile**

**$ do\_something < outputfile**

**$ do\_something 2> errfile**

**$ do\_something > all-output-file 2>&1**

**10. pipes**

**11. Searching files**

**- locate and find -> locate zip | grep bin**

**- wildcards.**

**Wildcards and Matching File Names**

**? -> Single character**

**\* -> matches any string of characters**

**[sed] -> Matches any characters in the set of characters,**

**[! sed] -> Matches any characters not in the set of characters.**

**12. find**

**$ find /usr -name gcc**

**$ find /usr -type d -name gcc**

**$ find /usr -type f -name gcc**