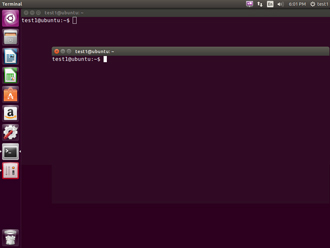
**Chapter 13: Manipulating Text**

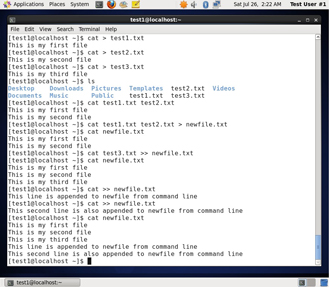
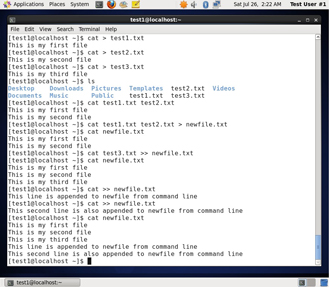
**Learning Objectives**



By the end of this chapter, you should be able to:

* Display and append to file contents using **cat** and **echo.**
* Edit and print file contents using **sed** and **awk**.
* Search for patterns using**grep**.
* Use multiple other utilities for file and text manipulation.
* **Command Line Tools**
* [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_03a.jpg)
* 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 utiltities.
* **cat**
* **cat** is short for concatenate and is one of the most frequently used Linux command line utilties. It is often used to read and print files as well as for simply viewing file contents. To view a file, use the following command:  $ cat <filename>
* For example, cat readme.txt will display the contents of readme.txt on the terminal. Often the main purpose of **cat,**however, is to combine (concatenate) multiple files together. You can perform the actions listed in the following table using **cat**:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| cat file1 file2 | Concatenate multiple files and display the output; i.e., the entire content of the first file is followed by that of the second file. |
| cat file1 file2 > newfile | Combine multiple files and save the output into a new file. |
| cat file >> existingfile | Append a file to the end of an existing file. |
| cat > file | Any subsequent lines typed will go into the file until CTRL-D is typed. |
| cat >> file | Any subsequent lines are appended to the file until CTRL-D is typed. |

* The **tac** command (**cat** spelled backwards) prints the lines of a file in reverse order. (Each line remains the same but the order of lines is inverted.) The syntax of **tac**is exactly the same as for **cat** as in:
* $ tac file $ tac file1 file2 > newfile
* **Using cat Interactively**
* [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch012_screen5.jpg)
* **cat**can be used to read from standard input (such as the terminal window) if no files are specified. You can use the > operator to create and add lines into a new file, and the >> operator to append lines (or files) to an existing file.
* To create a new file, at the command prompt type cat > <filename> and press the **Enter** key.
* This command creates a new file and waits for the user to edit/enter the text. After you finish typing the required text, press **CTRL-D** at the beginning of the next line to save and exit the editing.
* Another way to create a file at the terminal is cat > <filename> << EOF. A new file is created and you can type the required input. To exit, enter EOF at the beginning of a line.
* Note that EOF is case sensitive. (One can also use another word, such as STOP.)
* **Using cat Interactively**
* [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch012_screen5.jpg)
* **cat**can be used to read from standard input (such as the terminal window) if no files are specified. You can use the > operator to create and add lines into a new file, and the >> operator to append lines (or files) to an existing file.
* To create a new file, at the command prompt type cat > <filename> and press the **Enter** key.
* This command creates a new file and waits for the user to edit/enter the text. After you finish typing the required text, press **CTRL-D** at the beginning of the next line to save and exit the editing.
* Another way to create a file at the terminal is cat > <filename> << EOF. A new file is created and you can type the required input. To exit, enter EOF at the beginning of a line.
* Note that EOF is case sensitive. (One can also use another word, such as STOP.)
* **Note: The next few screens cover the demonstration and Try-It-Yourself activity. You can view a demonstration and practice the procedure through the Try-It-Yourself activity.**

**echo**

**echo** simply displays (echoes) text. It is used simply as in:

$ echo string

**echo** can be used to display a string on **standard output** (i.e., the terminal) or to place in a new file (using the > operator) or append to an already existing file (using the >> operator).

The –e option along with the following switches is used to enable special character sequences, such as the **newline** character or horizontal **tab**.

* \n  represents newline
* \t  represents horizontal tab

**echo**is particularly useful for viewing the values of environment variables (built-in shell variables). For example, echo $USERNAME will print the name of the user who has logged into the current terminal.

The following table lists **echo** commands and their usage:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| echo string > newfile | The specified string is placed in a new file. |
| echo string >> existingfile | The specified string is appended to the end of an already existing file. |
| echo $variable | The contents of the specified environment variable are displayed. |

**Try-It-Yourself: Using** echo

Tasks to be performed:

* Using echo, display a string: "This is a test.". [Note: Do not change the case and add additional characters like quotes, period and comma]

View the value of environmental variable $SHELL.

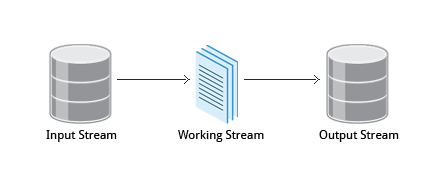
**Introduction to sed and awk**



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 more comprehensive language utilities such as **python** and **perl**, rather than use**sed** and **awk** (and some other utilities we'll 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 excecute faster. There are times (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, usually placing the contents into a new file. Its name is an abbreviation for **stream editor**.

**sed**can filter text as well as perform substitutions in data streams, working like a churn-mill.

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 following table:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| 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. |

The -e command option allows you to specify multiple editing commands simultaneously at the command line.

**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:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| sed s/pattern/replace\_string/ file | Substitute first string occurrence in a line |
| sed s/pattern/replace\_string/g file | Substitute all string occurrences in a 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 file > 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:

* It is a powerful utility and interpreted programming language.
* It is used to manipulate data files, retrieving, and processing text.
* 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:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| awk ‘command’ var=value file | Specify a command directly at the command line |
| awk -f scriptfile var=value file | Specify a file that contains the script to be executed along with f |

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**

**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:

* It is a powerful utility and interpreted programming language.
* It is used to manipulate data files, retrieving, and processing text.
* 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:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| awk ‘command’ var=value file | Specify a command directly at the command line |
| awk -f scriptfile var=value file | Specify a file that contains the script to be executed along with f |

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 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:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| 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 $6 }' /etc/passwd | Print first and sixth field of every line |

**File Manipulation Utilities**

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

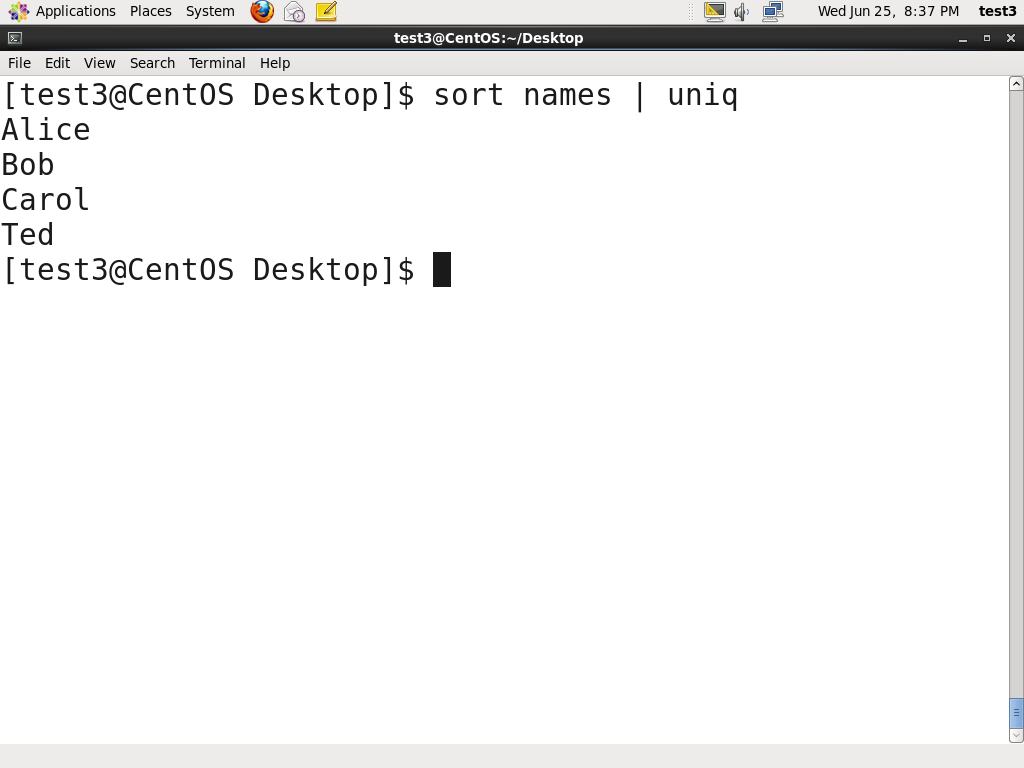
* sort
* uniq
* paste
* join
* split
* **sort**
* **sort**is used to rearrange the lines of a text file either in ascending or descending order, according to a sort key. You can also sort by particular fields of a file. The default sort key is the order of the ASCII characters (i.e., essentially alphabetically).
* **sort** can be used as follows:

|  |  |
| --- | --- |
| **Syntax** | **Usage** |
| sort <filename> | Sort the lines in the specified file |
| cat file1 file2 | sort | Append the two files, then sort the lines and display the output on the terminal |
| sort -r <filename> | Sort the lines in reverse order |

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**.

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_Chapter12_Screen24_B_.jpg)[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_Chapter12_Screen24_A_.jpg)

**uniq**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_25a.jpg)

**uniq**is used to remove duplicate lines in a text file and is useful for simplifying text display. **uniq**requires that the duplicate entries to be removed are consecutive. Therefore one often runs **sort**first and then pipes the output into **uniq**;if **sort** is passed the -u option it can do it all this in one step. In the example shown, the file is called names and was originally Ted, Bob, Alice, Bob, Carol, Alice.

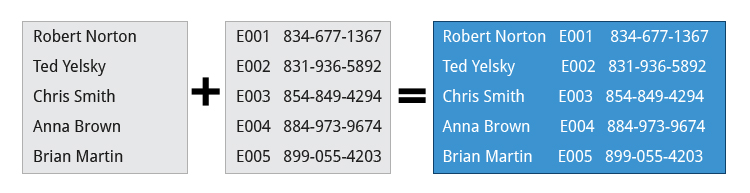
To remove duplicate entries from some files, use the following command: sort file1 file2 | uniq > file3

**OR**

sort -u file1 file2 > file3

To count the number of duplicate entries, use the following command: uniq -c filename

**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?

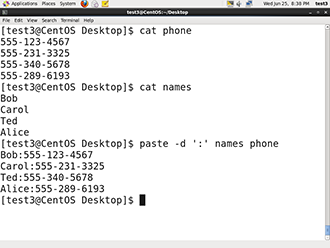
**paste** can be used to create a single file containing all three columns. The different columns are identified based on delimiters (spacing used to separate two fields). For example, delimiters can be a blank space, a tab, or an **Enter**. In the image provided, a single space is used as the delimiter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

**Using paste**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_28a.jpg)

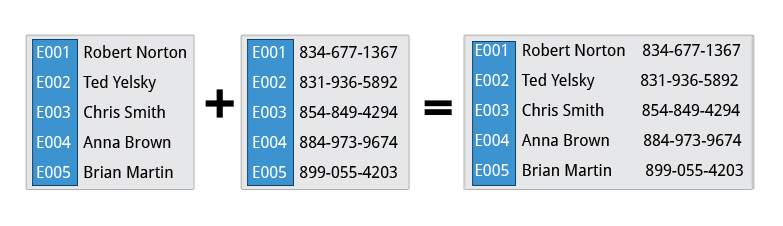
**paste** can be used to combine fields (such as name or phone number) from different files as well as combine lines from multiple files. For example, line one from file1 can be combined with line one of file2, line two from file1 can be combined with line two of file2, and so on.

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.

**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.

**Using join**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen30.jpg)

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

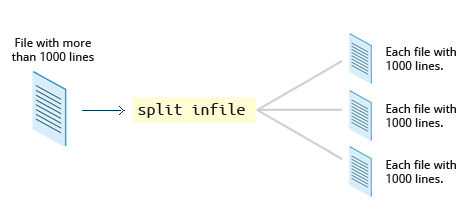
For example, the common field (i.e., it contains the same values) among the phonebook and directory files is the phone number, as shown by the output of the following **cat**commands:

$ cat phonebook 555-123-4567 Bob 555-231-3325 Carol 555-340-5678 Ted 555-289-6193 Alice

$ cat directory 555-123-4567 Anytown 555-231-3325 Mytown 555-340-5678 Yourtown 555-289-6193 Youngstown

The result of **join**ing these two file is as shown in the output of the following command: $ join phonebook directory 555-123-4567 Bob Anytown 555-231-3325 Carol Mytown 555-340-5678 Ted Yourtown 555-289-6193 Alice Youngstown

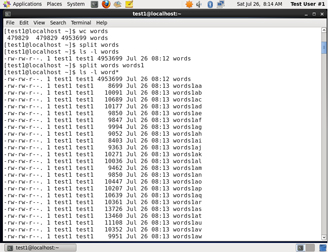
**split**



**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 1,000-line segments. The original file remains unchanged, and 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>.

**Using split**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch012_screen32.jpg)

To demonstrate the use of **split,** we'll apply it to an american-english dictionary file of over 99,000 lines:

$ wc -l american-english 99171 american-english

where we have used the **wc**program (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 equal-sized segments named 'dictionary'.

$ ls -l dictionary\* -rw-rw-r 1 me me 8552 Mar 23 20:19 dictionaryab -rw-rw-r 1 me me 8653 Mar 23 20:19 dictionaryaa . . .

**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 **metacharacters**, 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; we'll only skim the surface here.

These regular expressions are different from the wildcards (or "metacharacters") used in filename matching in command shells such as **bash** (which were covered in the earlier Chapter on Command Line Operations)**.** The table lists search patterns and their usage.

|  |  |
| --- | --- |
| **Search Patterns** | **Usage** |
| .(dot) | Match any single character |
| a|z | Match a or z |
| $ | Match end of string |
| \* | 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:

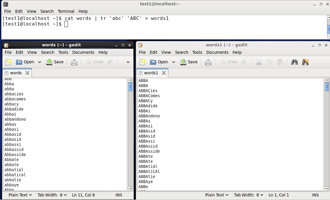
|  |  |
| --- | --- |
| **Command** | **Usage** |
| a.. | matches azy |
| b.|j. | matches both br and ju |
| ..$ | matches og |
| l.\* | matches lazy dog |
| l.\*y | matches lazy |
| the.\* | matches the whole sentence |

**grep**

**grep**is an 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.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| 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. |

**tr**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_42a.jpg)

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

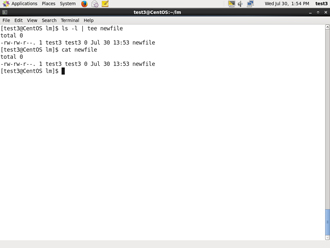
$ tr [options] set1 [set2]

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 uppercase characters to lowercase, at the command prompt type cat city | tr a-z A-Z and press the **Enter** key.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| $ 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 |
| $ tr -cd [:print:] < file.txt | Remove all non-printable character from a file |
| $ tr -s '\n' ' ' < file.txt | Join all the lines in a file into a single line |

**tee**

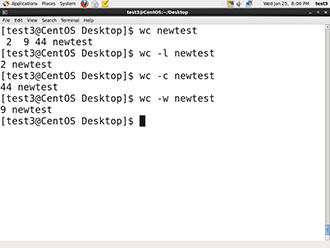
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_43a.jpg)

**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**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_44a.jpg)

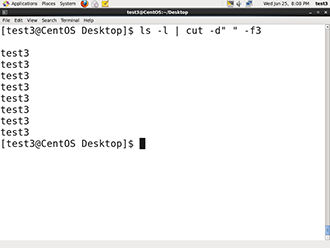
**wc** (word count) counts the number of lines, words, and characters in a file or list of files. Options are given in the table below.

By default all three of these options are active.

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

|  |  |
| --- | --- |
| **Option** | **Description** |
| –l | display the number of lines. |
| -c | display the number of characters. |
| -w | display the number of words. |

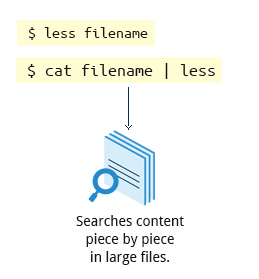
**cut**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_45a.jpg)

**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.

**Working with Large Files**



System administrators need to work with configuration files, text files, documentation files, and log files. Some of these file 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.

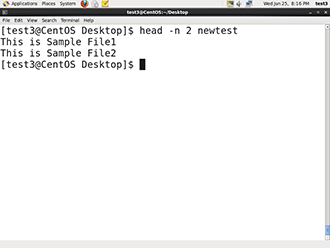
For example, a banking system might maintain one simple large log file to record details of all of one day's ATM transactions. Due to a security attack or a malfunction, the administrator might be forced to check for some data by navigating within the file. In such cases, directly opening the file in 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 the file can be done by typing either of the two following commands:

$ less <filename> $ cat <filename> | less

By default, manual (i.e., the **man** command) pages are sent through the **less** command.

**head**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_51a.jpg)

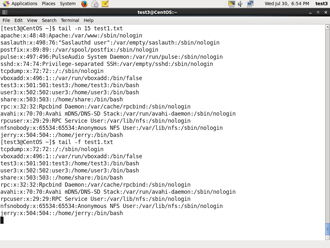
**head**reads the first few lines of each named file (10 by default) and displays it on standard output. You can give a different number of lines in an option.

For example, If you want to print the first 5 lines from amtrans.txt, use the following command:

$ head –n 5 atmtrans.txt

(You can also just say head -5 atmtrans.txt.)

**tail**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_52a.jpg)

**tail**prints the last few lines of each named file (10 by default) 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 amtrans.txt, use the following command:

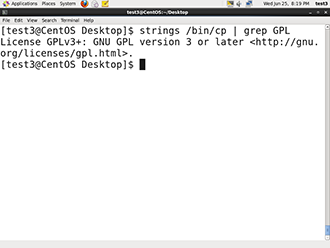
$ tail -n 15 atmtrans.txt

(You can also just say tail -15 atmtrans.txt.) To continually monitor new output in a growing log file:

$ tail -f atmtrans.txt

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

**strings**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_53a.jpg)

**strings**is 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 z Command Family**

When working with compressed files many standard commands can not be used directly. For many commonly-used file and text manipulation programs there is also a version especially designed to work directly with compressed files. These associated utilities have the letter **z** prefixed to their name. For example, we have utility programs such as **zcat**, **zless**, **zdiff,** and **zgrep**.

Here is a table listing some s z family commands:

|  |  |
| --- | --- |
| **Command** | **Description** |
| $ zcat compressed-file.txt.gz | To view a compressed file |
| $ zless <filename>.gz  or  $ zmore <filename>.gz | To page through a compressed file |
| $ zgrep -i less test-file.txt.gz | To search inside a compressed file |
| $ zdiff filename1.txt.gz  filename2.txt.gz | To compare two compressed files |

Note that if you run **zless** on an uncompressed file, it will still work and ignore the decompression stage. There are also equivalent utility programs for other compression methods besides **gzip**; i.e, we have **bzcat** and **bzless** associated with **bzip2,**and **xzcat** and **xzless**associated with **xz**.

**Try-It-Yourself: Using** head **and** tail

Tasks to be performed:

* View the first five lines of the **newtest** file using the head.

View the last five lines of the **newtest** file using the tail.

**Summary (1 of 3)**



You have completed this chapter. 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 a 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.

**Summary (2 of 3)**



* **uniq** eliminates duplicate entries in a text file.
* **paste**combines fields from different files and 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.

**Summary (3 of 3)**



* **tee**accepts 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.

**Chapter 14: Printing**

**Learning Objectives**



By the end of this chapter, you should know how to:

* Configure a printer on a Linux machine
* Print documents.

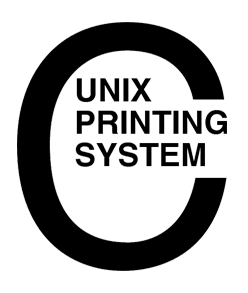
Manipulate postscript and pdf files using command line utilities.

**Introduction to Printing**



To manage printers and print directly from a computer or across a networked environment, you need to know how to configure and install a printer. Printing itself requires software that converts information from the application you are using to a language your printer can understand. The Linux standard for printing software is the **Common UNIX Printing System (CUPS)**.

**CUPS Overview**

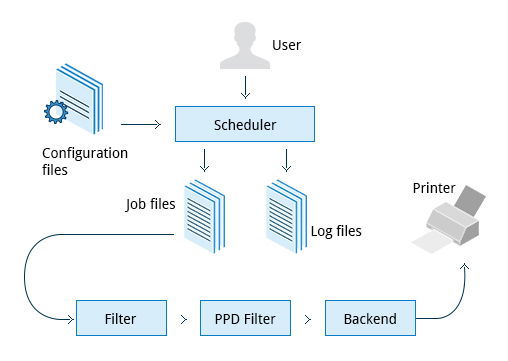


**CUPS** is the software that is used behind the scenes to print from applications like a web browser or **LibreOffice**. It converts page descriptions produced by your application (put a paragraph here, draw a line there, and so forth) and then sends the information to the printer. It acts as a **print server** for local as well as network printers.

Printers manufactured by different companies may use their own particular print languages and formats. **CUPS** uses a modular printing system which accommodates a wide variety of printers and also processes various data formats. This makes the printing process simpler; you can concentrate more on printing and less on how to print.

Generally, the only time you should need to configure your printer is when you use it for the first time. In fact, **CUPS** often figures things out on its own by detecting and configuring any printers it locates.

**How Does CUPS Work?**

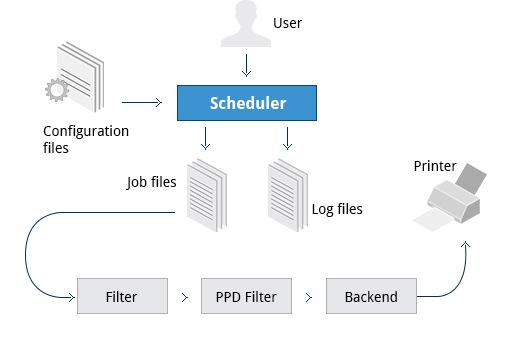


**CUPS** carries out the printing process with the help of its various components:

* Configuration Files
* Scheduler
* Job Files
* Log Files
* Filter
* Printer Drivers
* Backend

You will learn about each of these components in detail in the next few screens.

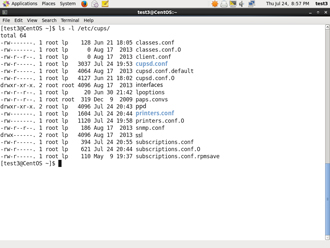
**Scheduler**



**CUPS** is designed around a **print scheduler** that manages print jobs, handles administrative commands, allows users to query the printer status, and manges the flow of data through all **CUPS** components.

As you'll see shortly, CUPS has a browser-based interface which allows you to view and manipulate the order and status of pending print jobs.

**Configuration Files**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch13_screen_07.jpg)

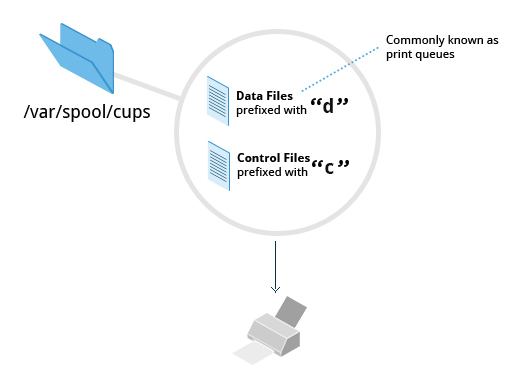
The print scheduler reads server settings from several configuration files, the two most important of which are cupsd.conf and printers.conf. These and all other **CUPS** related configuration files are stored under the /etc/cups/ directory.

cupsd.conf is where most system-wide settings are located; it does not contain any printer-specific details. Most of the settings available in this file relate to network security, i.e. which systems can access **CUPS** network capabilities, how printers are advertised on the local network, what management features are offered, and so on.

printers.conf is where you will find the printer-specific settings. For every printer connected to the system, a corresponding section describes the printer’s status and capabilities. This file is generated only after adding a printer to the system and should not be modified by hand.

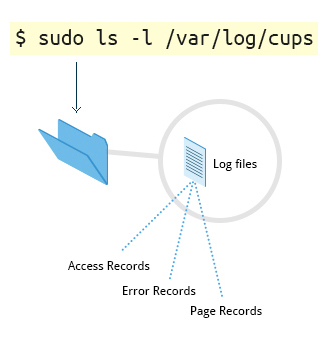
You can view the full list of configuration files by typing: ls -l /etc/cups/

**Job Files**



**CUPS** stores print requests as files under the /var/spool/cups directory (these can actually be accessed before a document is sent to a printer). Data files are prefixed with the letter **d** while control files are prefixed with the letter **c**. After a printer successfully handles a job, data files are automatically removed. These data files belong to what is commonly known asthe**print queue**.

**Log Files**

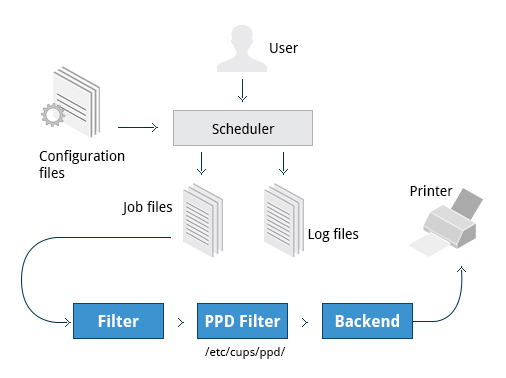


Log files are placed in /var/log/cups and are used by the scheduler to record activities that have taken place. These files include access, error, and page records.

To view what log files exist, type:  sudo ls -l /var/log/cups

(Note on some distributions permissions are set such that you don't need the **sudo**.) You can view the log files with the usual tools.

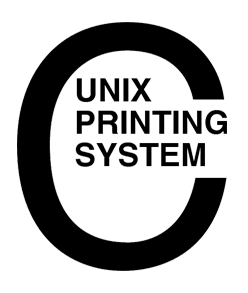
**Filters, Printer Drivers, and Backends**



**CUPS** uses **filters** to convert job file formats to printable formats. Printer **drivers** contain descriptions for currently connected and configured printers, and are usually stored under /etc/cups/ppd/. The print data is then sent to the printer through a filter and via a **backend** that helps to locate devices connected to the system.

So In short, when you execute a print command, the scheduler validates the command and processes the print job creating job files according to the settings specified in configuration files. Simultaneously, the scheduler records activities in the log files. Job files are processed with the help of the filter, printer driver, and backend, and then sent to the printer.

**Installing CUPS**



Due to printing being a relatively important and fundamental feature of any Linux distribution, most Linux systems come with **CUPS** preinstalled. In some cases, especially for Linux server setups, **CUPS** may have been left uninstalled. This may be fixed by installing the corresponding package manually. To install **CUPS**, please ensure that your system is connected to the Internet.

**Demonstration of Installing CUPS**

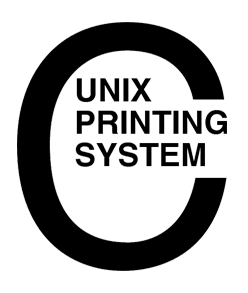
You can use the commands shown below to manually install **CUPS**:

**- CentOS:**$ sudo yum install cups **- OpenSUSE:**$ sudo zypper install cups **- Ubuntu:**$ sudo apt-get install cups

The video below demonstrates this procedure for **Ubuntu**, the other two are similar, once the correct install command is provided.

**Note:** **CUPS** features are also supported by other packages such as cups-common and libcups2, which contains the core **CUPS** libraries. The above install command will make sure any needed packages are also installed.

**Managing CUPS**



After installing **CUPS**, you'll need to start and manage the **CUPS** daemon so that **CUPS** is ready for configuring a printer. Managing the **CUPS** daemon is simple; all management features are wrapped around the **cups** init script, which can be easily started, stopped, and restarted.

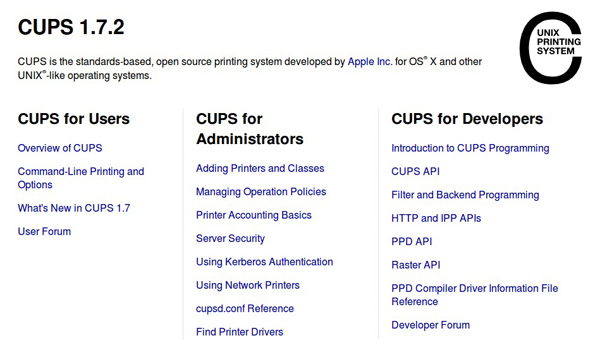
**Configuring a Printer from the GUI**

Each Linux distribution has a GUI application that lets you add, remove, and configure local or remote printers. Using this application, you can easily set up the system to use both local and network printers. The following screens show how to find and use the appropriate application in each of the distribution families covered in this course.

When configuring a printer, make sure the device is currently turned on and connected to the system; if so it should show up in the printer selection menu. If the printer is not visible, you may want to troubleshoot using tools that will determine if the printer is connected. For common USB printers, for example, the **lsusb** utility will show a line for the printer. Some printer manufacturers also require some extra software to be installed in order to make the printer visible to **CUPS**, however, due to the standardization these days, this is rarely required.

**Adding Printers from the CUPS Web Interface**

A fact that few people know is that **CUPS** also comes with its own web server, which makes a configuration interface available via a set of CGI scripts.



This web interface allows you to:

* 1. Add and remove local/remote printers
  2. Configure printers:
  + – Local/remote printers
  + – Share a printer as a CUPS server
  + Control print jobs:
  + – Monitor jobs
  + – Show completed or pending jobs
  + – Cancel or move jobs

The **CUPS** web interface is available on your browser at: [http://localhost:631](http://localhost:631/)

Some pages require a username and password to perform certain actions, for example to add a printer. For most Linux distributions, you must use the root password to add, modify, or delete printers or classes.

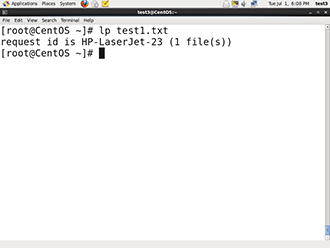
**Printing from the Graphical Interface**

Many graphical applications allow users to access printing features using the **CTRL-P** shortcut. To print a file, you first need to specify the printer (or a file name and location if you are printing to a file instead) you want to use; and then select the page setup, quality, and color options. After selecting the required options, you can submit the document for printing. The document is then submitted to **CUPS**. You can use your browser to access the **CUPS** web interface at <http://localhost:631/> to monitor the status of the printing job. Now that you have configured the printer, you can print using either the Graphical or Command Line interfaces.

The screenshots below show the GUI interfaces for **CTRL-P** for (from left to right) **CentOS**, **Ubuntu** and **openSUSE**.

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Printing from the Command-Line Interface**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_Chapter13_Screen37.jpg)

**CUPS** provides two command-line interfaces, descended from the **System V** and **BSD**flavors of UNIX. This means that you can use either **lp** (System V) or **lpr** (BSD) to print. You can use these commands to print text, PostScript, PDF, and image files.

These commands are useful in cases where printing operations must be automated (from shell scripts, for instance, which contain multiple commands in one file). You will learn more about the shell scripts in the upcoming chapters on **bash**scripts.

**lp**is just a command line front-end to the **lpr** utility that passes input to **lpr**. Thus, we will discuss only **lp** in detail. In the example shown here, the task is to print the file called test1.txt.

**Using lp**

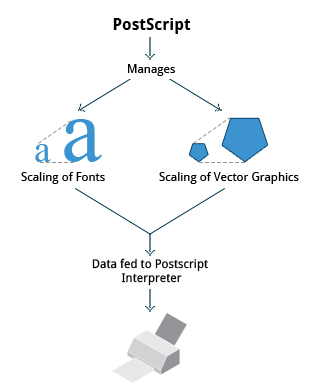
**lp**and **lpr** acceptcommand line options that help you perform all operations that the GUI can accomplish. **lp**is typically used with a file name as an argument.

Some  **lp** commands and other printing utitilies you can use are listed in the table.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| lp <filename> | To print the file to default printer |
| lp -d printer <filename> | To print to a specific printer (useful if multiple printers are available) |
| program | lp  echo string | lp | To print the output of a program |
| lp -n number <filename> | To print multiple copies |
| lpoptions -d printer | To set the default printer |
| lpq -a | To show the queue status |
| lpadmin | To configure printer queues |

The **lpoptions** utility can be used to set printer options and defaults. Each printer has a set of **tags** associated with it, such as the default number of copies and authentication requirements. You can execute the command lpoptions help to obtain a list of supported options. **lpoptions**can also be used to set system-wide values, such as the default printer.

**Working with PostScript**



**PostScript** is a standard **page description language.** It effectively manages scaling of fonts and vector graphics to provide quality printouts. It is purely a text format that contains the data fed to a PostScript interpreter. The format itself is a language that was developed by **Adobe** in the early 1980s to enable the transfer of data to printers.

Features of PostScript are:

* It can be used on any printer that is PostScript-compatible; i.e., any modern printer
* Any program that understands the PostScript specification can print to it

Information about page appearance, etc. is embedded in the page

**Working with enscript**

**enscript** is a tool that is used to convert a text file to PostScript and other formats. It also supports **Rich Text Format (RTF)** and **HyperText Markup Language (HTML)**. For example, you can convert a text file to two column (-2) formatted **PostScript** using the command: enscript -2 -r -p psfile.ps textfile.txt. This command will also rotate (-r) the output to print so the width of the paper is greater than the height (aka landscape mode) thereby reducing the number of pages required for printing.

The commands that can be used with **enscript** are listed in the table below (for a file called 'textfile.txt').

|  |  |
| --- | --- |
| **Command** | **Usage** |
| enscript -p psfile.ps textfile.txt | Convert a text file to PostScript (saved to psfile.ps) |
| enscript -n -p psfile.ps textfile.txt | Convert a text file to n columns where n=1-9 (saved in psfile.ps) |
| enscript textfile.txt | Print a text file directly to the default printer |

**Viewing PDF Content**



Linux has many standard programs that can read PDF files as well as many applications that can easily create them, including all available office suites such as **LibreOffice.**

The most common Linux PDF readers are:

* **Evince** is available on virtually all distributions and the most widely used program.
* **Okular** is based on the older **kpdf** and available on any distribution that provides the **KDE** environment.
* **GhostView** is one of the first open source PDF readers and is universally available.
* **Xpdf** is one of the oldest open source PDF readers and still has a good user base.

All of these open source PDF readers support and can read files following the PostScript standard unlike the proprietary **Adobe Acrobat Reader**, which was once widely used on Linux systems but with the growth of these excellent programs, very few Linux users use it today.

**Modifying PDFs with pdftk**

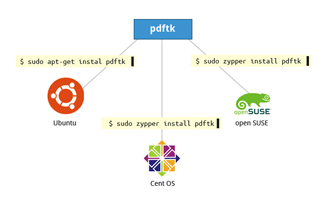


At times, you may want to merge, split, or rotate PDF files; not all of these operations can be achieved while using a PDF viewer.  A great way to do this is to use the "PDF Toolkit", **pdftk**, to perform a very large variety of sophisticated tasks. Some of these operations include:

* Merging/Splitting/Rotating PDF documents
* Repairing corrupted PDF pages
* Pulling single pages from a file
* Encrypting and decrypting PDF files
* Adding, updating, and exporting a PDF’s **metadata**
* Exporting bookmarks to a text file
* Filling out PDF forms

In short, there’s very little **pdftk**cannot do when it comes to working with PDF files; it is indeed the Swiss Army knife of PDF tools.

**Installing pdftk on Different Family Systems**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch13_Screen_48.jpg)

To install **pdftk** on **Ubuntu**, use the following command:

$ sudo apt-get install pdftk

On **CentOS:**

$ sudo yum install pdftk

On **openSUSE**:

$ sudo zypper install pdftk

You may find that **CentOS**(and **RHEL**) don't have **pdftk** in their packaging system, but you can obtain the PDF Toolkit directly from the PDF Lab’s website by downloading from: <http://www.pdflabs.com/docs/install-pdftk-on-redhat-or-centos/>

Click the image to view an enlarged version.

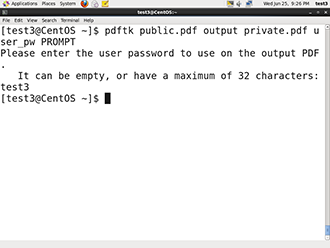
**Using pdftk**

You can accomplish a wide variety of tasks using **pdftk** including:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| pdftk 1.pdf 2.pdf cat output 12.pdf | Merge the two documents 1.pdf and 2.pdf. The output will be saved to 12.pdf. |
| pdftk A=1.pdf cat A1-2 output new.pdf | Write only pages 1 and 2 of 1.pdf. The output will be saved to new.pdf. |
| pdftk A=1.pdf cat A1-endright output new.pdf | Rotate all pages of 1.pdf 90 degrees clockwise and save result in new.pdf**.** |

|  |  |
| --- | --- |
| **Command** | **Usage** |
| pdftk 1.pdf 2.pdf cat output 12.pdf | Merge the two documents 1.pdf and 2.pdf. The output will be saved to 12.pdf. |
| pdftk A=1.pdf cat A1-2 output new.pdf | Write only pages 1 and 2 of 1.pdf. The output will be saved to new.pdf. |
| pdftk A=1.pdf cat A1-endright output new.pdf | Rotate all pages of 1.pdf 90 degrees clockwise and save result in new.pdf**.** |

**Encrypting PDF Files**

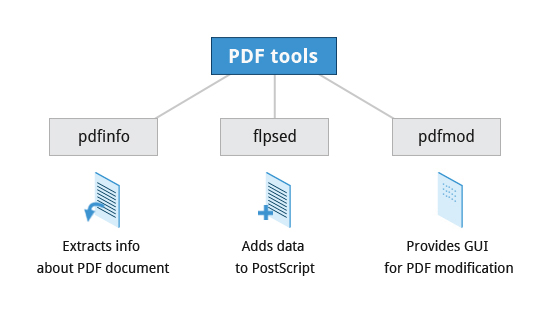
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch13_screen_50.jpg)

If you’re working with PDF files that contain confidential information and you want to ensure that only certain people can view the PDF file, you can apply a password to it using the user\_pw option. One can do this by issuing a command such as:

$ pdftk public.pdf output private.pdf user\_pw PROMPT

When you run this command, you will receive a prompt to set the required password, which can have a maximum of 32 characters. A new file, private.pdf, will be created with the identical content as public.pdf, but anyone will need to type the password to be able to view it.

**Using Additional Tools**



You can use other tools, such as **pdfinfo**, **flpsed**, and **pdfmod** to work with PDF files.

**pdfinfo** can extract information about PDF files, especially when the files are very large or when a graphical interface is not available.

**flpsed** can add data to a PostScript document. This tool is specifically useful for filling in forms or adding short comments into the document.

**pdfmod** is a simple application that provides a graphical interface for modifying PDF documents. Using this tool, you can reorder, rotate, and remove pages; export images from a document; edit the title, subject, and author; add keywords; and combine documents using drag-and-drop action.

For example, to collect the details of a document, you can use the following command: $ pdfinfo /usr/share/doc/readme.pdf

**Converting Between PostScript and PDF**

Most users today are far more accustomed to working with files in PDF format, viewing them easily either on the Internet through their browser or locally on their machine. The PostScript format is still important for various technical reasons that the general user will rarely have to deal with.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| pdf2ps file.pdf | Converts file.pdf tofile.ps |
| ps2pdf file.ps | Convertsfile.ps tofile.pdf |
| pstopdf input.ps output.pdf | Converts input.psto output.pdf |
| pdftops input.pdf output.pdf | Converts input.pdf to output.ps |

From time to time you may need to convert files from one format to the other, and there are very simple utilities for accomplishing that task. **ps2pdf** and **pdf2ps** are part of the **ghostscript**package installed on or available on all Linux distributions. As an alternative, there **pstopdf** and **pdftops** which are usually part of the **poppler** package. Unless you are doing a lot of conversions or need some of the fancier options (which you can read about in the **man pages** for these utilties) it really doesn't matter which ones you use.

Some usage examples:

**Summary (1 of 2)**



You have completed this chapter. Let’s summarize the key concepts covered:

* **CUPS** provides two command-line interfaces: the **System V** and **BSD** interfaces.
* The CUPS interface is available at [http://localhost:631](http://localhost:631/)
* **lp** and **lpr**  are used to submit a document to **CUPS** directly from the command line.
* **lpoptions** can be used to set printer options and defaults.
* PostScript effectively manages scaling of fonts and vector graphics to provide quality prints.
* **enscript** is used to convert a text file to PostScript and other formats.

**Summary (2 of 2)**



* **Portable Document Format (PDF)** is the standard format used to exchange documents while ensuring a certain level of consistency in the way the documents are viewed.
* **pdftk** joins and splits PDFs; pulls single pages from a file; encrypts and decrypts PDF files; adds, updates, and exports a PDF’s metadata; exports bookmarks to a text file; adds or removes attachments to a PDF; fixes a damaged PDF; and fills out PDF forms.
* **pdfinfo** can extract information about PDF documents.
* **flpsed** can add data to a PostScript document.
* **pdfmod** is a simple application with a graphical interface that you can use to modify PDF documents.

**Chapter 15:**

**Bash Shell Scripting**

**Learning Objectives**

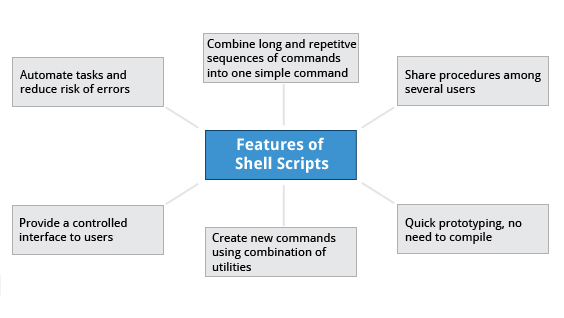


By the end of this chapter, you should be able to:

* Explain the features and capabilities of **bash** **shell scripting**.
* Know the basic syntax of scripting statements.
* Be familiar with various methods and constructs used.
* Know how to test for properties and existence of files and other objects.
* Use conditional statements, such as **if-then-else** blocks.

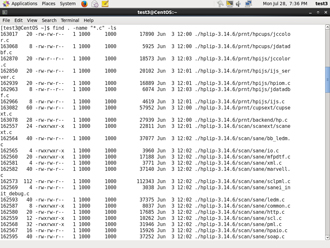
Perform arithmetic operations using scripting language.

**Introduction to Scripts**



Suppose you want to look up a filename, check if the associated file exists, and then respond accordingly, displaying a message confirming or not confirming the file's existence. If you only need to do it once, you can just type a sequence of commands at a terminal. However, if you need to do this multiple times, automation is the way to go. In order to automate sets of commands you’ll need to learn how to write **shell scripts**, the most common of which are used with **bash**. The graphic illustrates several of the benefits of deploying scripts.

**Introduction to Shell Scripts**

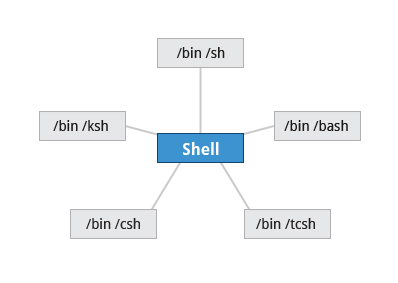
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen04a.jpg)

Remember from our earlier discussion, a **shell** is a command line **interpreter** which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in. For example typing: find . -name "\*.c" -ls at the command line accomplishes the same thing as executing a script file containing the lines:

#!/bin/bash find . -name "\*.c" -ls

The #!/bin/bash in the first line should be recognized by anyone who has developed any kind of script in UNIX environments. The first line of the script, that starts with #!, contains the full path of the command interpreter (in this case /bin/bash) that is to be used on the file. As we will see on the next screen, you have a few choices depending upon which scripting language you use.

**Command Shell Choices**



The command **interpreter** is tasked with executing statements that follow it in the script. Commonly used interpreters include: /usr/bin/perl, /bin/bash, /bin/csh, /usr/bin/python and /bin/sh.

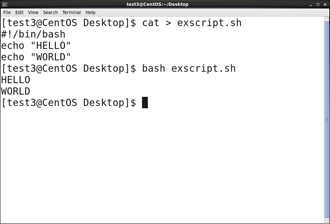
Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command-line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with several users.

Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

/bin/sh /bin/bash /bin/tcsh /bin/csh /bin/ksh

Most Linux users use the default **bash** shell, but those with long UNIX backgrounds with other shells may want to override the default.

**bash Scripts**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen06a.jpg)

Let's write a simple **bash** script that displays a two-line message on the screen. Either type

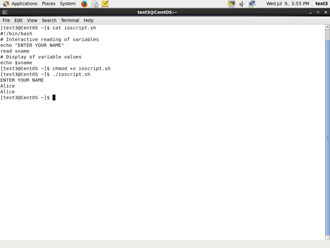
$ cat > exscript.sh   #!/bin/bash   echo "HELLO"   echo "WORLD"

and press **ENTER** and **CTRL-D** to save the file, or just create exscript.sh in your favorite text editor. Then, type chmod +x exscript.sh to make the file executable. (The chmod +x command makes the file executable for all users.) You can then run it by simply typing ./exscript.sh or by doing:

$ bash exscript.sh   HELLO   WORLD

Note if you use the second form, you don't have to make the file executable.

**Interactive Example Using bash Scripts**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/ch14_screen7.jpg)

Now, let's see how to create a more interactive example using a **bash** script. The user will be prompted to enter a value, which is then displayed on the screen. The value is stored in a temporary variable, sname. We can reference the value of a shell variable by using a $ in front of the variable name, such as $sname. To create this script, you need to create a file named ioscript.sh in your favorite editor with the following content:

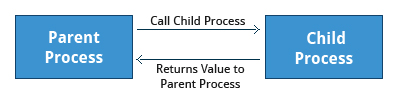
   #!/bin/bash    # Interactive reading of variables    echo "ENTER YOUR NAME"    read sname    # Display of variable values    echo $sname

Once again, make it executable by doing chmod +x ioscript.sh.

In the above example, when the script ./ioscript.sh is executed, the user will receive a prompt ENTER YOUR NAME. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.

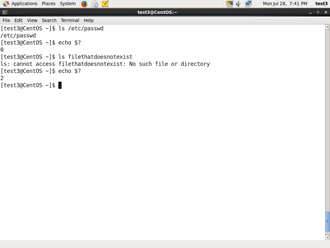
Additional note: The hash-tag/pound-sign/number-sign (#) is used to start comments in the script and can be placed anywhere in the line (the rest of the line is considered a comment).

**Return Values**



All shell scripts generate a **return value** upon finishing execution; the value can be set with the exit statement. Return values permit a process to monitor the exit state of another process often in a parent-child relationship. This helps to determine how this process terminated and take any appropriate steps necessary, contingent on success or failure.

**Viewing Return Values**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen08a.jpg)

As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value. An easy way to demonstrate success and failure completion is to execute **ls** on a file that exists and one that doesn't, as shown in the following example, where the return value is stored in the environment variable represented by $?:

$ ls /etc/passwd /etc/ passwd

$ echo $? 0

In this example, the system is able to locate the file /etc/passwd and returns a value of 0 to indicate success; the return value is always stored in the $? environment variable. Applications often translate these return values into meaningful messages easily understood by the user.

**Basic Syntax and Special Characters**

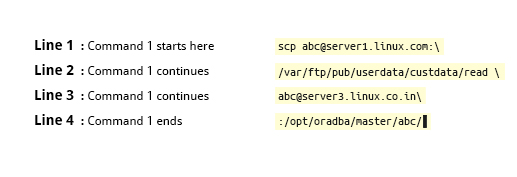
Scripts require you to follow a standard language **syntax**. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within **bash** scripts:

|  |  |
| --- | --- |
| **Character** | **Description** |
| # | Used to add a comment, **except** when used as \#, or as #! when starting a script |
| \ | Used at the end of a line to indicate continuation on to the next line |
| ; | Used to interpret what follows as a new command |
| $ | Indicates what follows is a variable |

Note that when # is inserted at the beginning of a line of commentary, the whole line is ignored.

# This line will not get executed.

**Splitting Long Commands Over Multiple Lines**



Users sometimes need to combine several commands and statements and even conditionally execute them based on the behaviour of operators used in between them. This method is called **chaining of commands**.

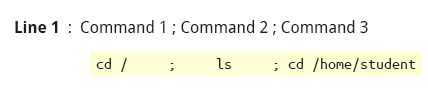
The **concatenation operator** (**\**) is used to concatenate large commands over several lines in the shell.

For example, you want to copy the file **/var/ftp/pub/userdata/custdata/read** from **server1.linux.com** to the **/opt/oradba/master/abc** directory on **server3.linux.co.in**. To perform this action, you can write the command using the \ operator as:

scp abc@server1.linux.com:\ /var/ftp/pub/userdata/custdata/read \ abc@server3.linux.co.in:\ /opt/oradba/master/abc/

The command is divided into multiple lines to make it look readable and easier to understand. The \ operator at the end of each line combines the commands from multiple lines and executes it as one single command.

**Putting Multiple Commands on a Single Line**



Sometimes you may want to group multiple commands on a single line. The ; (semicolon) character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.

The three commands in the following example will all execute even if the ones preceding them fail: $ make ; make install ; make clean

However, you may want to abort subsequent commands if one fails. You can do this using the && (and) operator as in:

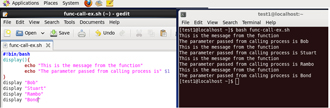
$ make && make install && make clean

If the first command fails the second one will never be executed. A final refinement is to use the || (or) operator as in:

$ cat file1 || cat file2 || cat file3

In this case, you proceed until something succeeds and then you stop executing any further steps.

**Functions**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen15a.jpg)

A **function** is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called **subroutines.** Using functions in scripts requires two steps:

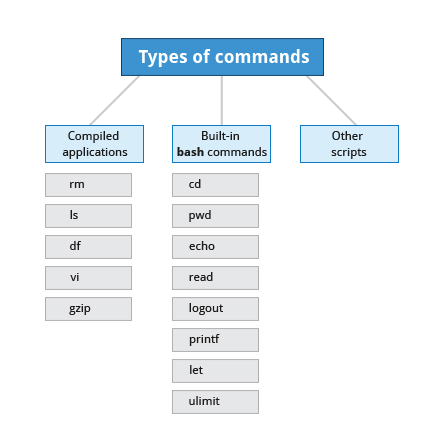
1. Declaring a function 2. Calling a function

The function declaration requires a name which is used to invoke it. The proper syntax is:      function\_name () {        command...     }

For example, the following function is named display:

    display () {        echo "This is a sample function"     }  The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary. In the full example shown in the figure, we are also showing an often-used refinement: how to pass an argument to the function.  The first argument can be referred to as $1, the second as $2, etc.

**Built-in Shell Commands**



Shell scripts are used to execute sequences of commands and other types of statements. Commands can be divided into the following categories:

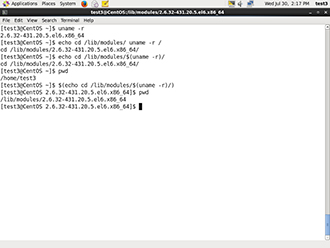
* Compiled applications
* Built-in **bash**commands
* Other scripts

Compiled applications are binary executable files that you can find on the filesystem. The shell script always has access to compiled applications such as **rm**, **ls**, **df**, **vi**, and **gzip**.

**bash** has many **built-in** commands which can only be used to display the output within a terminal shell or shell script. Sometimes these commands have the same name as executable programs on the system, such as **echo** which can lead to subtle problems. **bash** built-in commands include and cd, pwd, echo, read, logout, printf, let, and ulimit.

A complete list of **bash** built-in commands can be found in the **bash man** page, or by simply typing help.

**Command Substitution**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen17.jpg)

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

* By enclosing the inner command with backticks (`)
* By enclosing the inner command in $( )

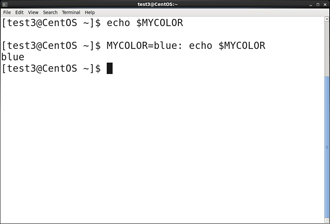
No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done.

Virtually any command can be executed this way. Both of these methods enable command subsitution; however, the $( ) method allows command nesting. New scripts should always use this more modern method.

For example:  $ cd /lib/modules/$(uname -r)/

In the above example, the output of the command "uname –r" becomes the argument for the cd command.

**Environment Variables**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen18a.jpg)

Almost all scripts use **variables** containing a value, which can be used anywhere in the script. These variables can either be user or system defined. Many applications use such **environment variables** (covered in the "User Environment" chapter) for supplying inputs, validation, and controlling behaviour.

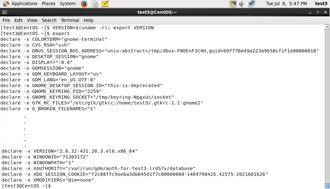
Some examples of standard environment variables are HOME, PATH, and HOST. When referenced, environment variables must be prefixed with the $ symbol as in $HOME. You can view and set the value of environment variables. For example, the following command displays the value stored in the PATH variable:

$ echo $PATH

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the MYCOLOR variable to blue:  $ MYCOLOR=blue

You can get a list of environment variables with the **env**, **set**, or **printenv**commands.

**Exporting Environment Variables**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen19.jpg)

By default, the environment variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be **exported** using the **export** statement as in:  export VAR=value  or  VAR=value ; export VAR

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

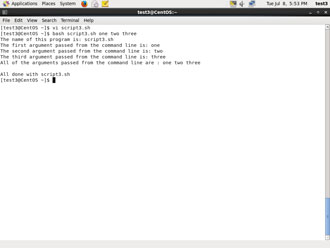
**Script Parameters**

Users often need to pass parameter values to a script, such as a filename, date, etc.  Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them.  These values can be text or numbers as in:

$ ./script.sh /tmp $ ./script.sh 100 200  Within a script, the parameter or an argument is represented with a $ and a number. The table lists some of these parameters.

|  |  |
| --- | --- |
| **Parameter** | **Meaning** |
| $0 | Script name |
| $1 | First parameter |
| $2, $3, etc. | Second, third parameter, etc. |
| $\* | All parameters |
| $# | Number of arguments |

**Using Script Parameters**

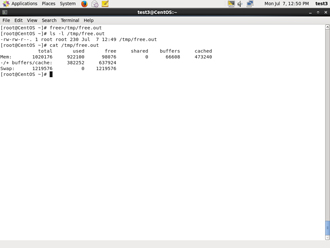
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen22.jpg)

Using your favorite text editor, create a new script file named **script3.sh** with the following contents:

#!/ bin/ bash echo The name of this program is: $0 echo The first argument passed from the command line is: $1 echo The second argument passed from the command line is: $2 echo The third argument passed from the command line is: $3 echo All of the arguments passed from the command line are : $\* echo echo All done with $0

Make the script executable with **chmod +x**.  Run the script giving it three arguments as in: script3.sh one two three, and the script is processed as follows:  $0 prints the script name: script3.sh $1 prints the first parameter: one $2 prints the second parameter: two $3 prints the third parameter: three $\* prints all parameters: one two three  The final statement becomes: All done with ./script3.sh

**Output Redirection**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen23a.jpg)

Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output **redirection**.

The > character is used to write output to a file. For example, the following command sends the output of **free**to the file /tmp/free.out:  $ free > /tmp/free.out

To check the contents of the /tmp/free.out file, at the command prompt type  cat /tmp/free.out.

Two > characters (>>) will append output to a file if it exists, and act just like > if the file does not already exist.

**Input Redirection**

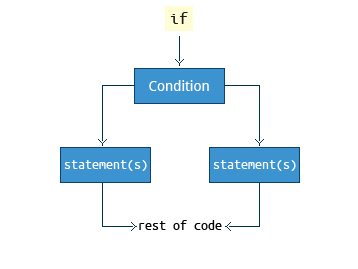
[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen24a.jpg)

Just as the output can be redirected **to** a file, the input of a command can be read **from** a file. The process of reading input from a file is called input redirection and uses the < character. If you create a file called script8.sh with the following contents:

#!/bin/bash echo “Word count” wc -l < /temp/free.out

and then execute it with chmod +x script8.sh ; ./script8.sh, it will count the number of lines from the **/temp/free.out** file and display the results.

**The if Statement**



Conditional decision making using an if statement, is a basic construct that any useful programming or scripting language must have.

When an if statement is used, the ensuing actions depend on the evaluation of specified conditions such as:

* Numerical or string comparisons
* Return value of a command (0 for success)
* File existence or permissions

In compact form, the syntax of an if statement is:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

A more general definition is:

if condition

then

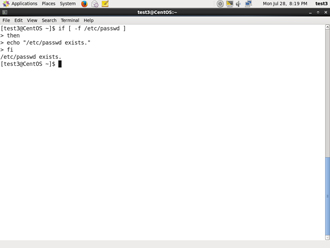
       statements

 else

statements

 fi

**Using the if Statement**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen30a.jpg)

The following if statement checks for the /etc/passwd file, and if the file is found it displays the message /etc/passwd exists.:

if [ -f /etc/passwd ] then     echo "/etc/passwd exists." fi

Notice the use of the square brackets ([]) to delineate the test condition. There are many other kinds of tests you can perform, such as checking  whether two numbers are equal to, greater than, or less than each other and make a decision accordingly; we will discuss these other tests.

In modern scripts you may see doubled brackets as in[[ -f /etc/passwd ]]. This is not an error. It is never wrong to do so and it avoids some subtle problems such as referring to an empty environment variable without surrounding it in double quotes; we won't talk about this here.

**Testing for Files**

You can  use the if statement to test for file attributes such as:

* File or directory existence
* Read or write permission
* Executable permission

For example, in the following example: if [ -f /etc/passwd ] ; then     ACTION fi

the if statement checks if the file /etc/passwd is a regular file. Note the very common practice of putting **“; then”** on the same line as the **if** statement.

**bash** provides a set of **file conditionals**, that can used with the if statement, including:

|  |  |
| --- | --- |
| **Condition** | **Meaning** |
| -e file | Check if the file exists. |
| -d file | Check if the file is a directory. |
| -f file | Check if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.) |
| -s file | Check if the file is of non-zero size. |
| -g file | Check if the file has sgid set. |
| -u file | Check if the file has suid set. |
| -r file | Check if the file is readable. |
| -w file | Check if the file is writable. |
| -x file | Check if the file is executable. |

You can view the full list of file conditions using the command man 1 test.

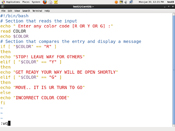
**Example of Testing of Strings**

You can use the if statement to compare strings using the operator == (two equal signs). The syntax is as follows:

if [ string1 == string2 ] ; then    ACTION fi

Let’s now consider an example of testing strings.

In the example illustrated here, the if statement is used to compare the input provided by the user and accordingly display the result.

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen35a.jpg)[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen35b.jpg)

**Numerical Tests**

You can use specially defined operators with the if statement to compare numbers. The various operators that are available are listed in the table.

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| -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 |

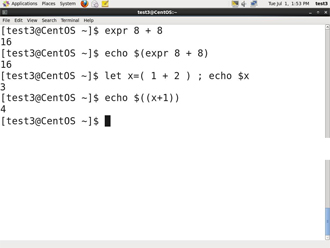
The syntax for comparing numbers is as follows: exp1 -op exp2

**Example of Testing for Numbers**

Let us now consider an example of comparing numbers using the various operators:

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen39a.jpg) [](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen39b.jpg)

**Arithmetic Expressions**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch14_screen44.jpg)

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

* Using the **expr** utility: **expr**is a standard but somewhat deprecated program. The syntax is as follows:  expr 8 + 8 echo $(expr 8 + 8)
* Using the $((...)) syntax: This is the built-in shell format. The syntax is as follows:  echo $((x+1))
* Using the built-in shell command let. The syntax is as follows:  let x=( 1 + 2 ); echo $x

In modern shell scripts the use of **expr** is better replaced with var=$((...))

**Summary (1 of 2)**



You have completed this chapter. Let’s summarize the key concepts covered:

* Scripts are a sequence of statements and commands stored in a file that can be executed, of which base is the most commonly used.
* Command substitution allows you to substitute the result of a command as a portion of another command.
* Functions or routines are a group of commands that are used for execution.

Environmental variables are quantities either pre-assigned by the shell or defined and modified by the user.

**Summary (2 of 2)**



* To make environment variables visible to child processes, they need to be **exported**.
* Scripts can behave differently based on the parameters (values) passed to them.
* The process of writing the output to a file is called output redirection.
* The process of reading input from a file is called input redirection.
* The if statement is used to select an action based on a condition.
* Arithmetic expressions consist of numbers and arithmetic operators, such as +, -, and \*.