



ABC IT Education

WE'LL TAKE YOU FROM ZERO TO HERO IN A SNAP

Linux Systems Administration

Linux Intermediate 1

Wildcards, I/O Redirection,
Comparing Files and Searching in Files

Wildcards

- In this wildcards lesson you are going to see what are, when to use them and how to use them.
- You will also see the various types of wildcards available in Linux.
- Practice make perfect so we are going to look at some examples dealing with wildcards.
- What is a wildcard? A wildcard is a character or a string (set of characters) used with commands to match file and directory names.
- Wildcards are used to create search patterns that when expanded will return a list of matching files and directories.
- Wildcards are also referred to as globs or glob patterns.

- 
- Globbing is the act of expanding a wildcard into the list of matching files and directories.
 - Wildcards can be used in conjunction with most Linux commands, especially if the command accepts a file or directory as an argument.
 - We can use a wildcard in this argument to specify a file or set of files.
 - For the most part, you will use wildcards with commands like 'ls', 'rm', 'cp', 'mv', and many others.
 - The 2 main wildcards are the asterisk (*) or star and the question mark (?).
 - When used, the asterisk will match zero or more characters, meaning it matches anything, while the question mark matches exactly 1 character.

- 
- **Examples** (filenames used are for example only)
 - To look for all files that end in '.txt', use '*.txt' to find all the files that end in '.txt'.
 - For a list of all files that start with the letter 'c', use 'c*'.
 - If you want to find all the files that start with an 'my' and end in '.txt', then use 'my*.txt'.
 - To find all the files that only have 1 character preceding '.txt', then use '?'.txt'.
 - To match all the 4 letter files that begin with an 'y', use the letter 'y' followed by 3 '?'.
 - To match all the files that start with 'b', and then are followed by exactly one additional character, and then end in '.txt', use 'b?.txt'.

More Wildcards

- In using wildcards, we can use a character class to create very specific search patterns.
- Character classes begin with the left bracket, then a list one or more characters you are interested in matching and then end with the right bracket.
- For example, if you wanted to match a one character long file name that is a vowel, you would use `[aeiou]`.
- If you wanted to match files that start with '9ja' followed by either an 'm' or a 'w', followed by zero or more characters, then use `'9ja[mw]*'`. And it will match files named, 9jaman, 9jawoman, 9jamusic, 9jam, 9jaw etc.
- To exclude characters in a match, use an exclamation mark (!), e.g. if you wanted to find all the files that do not start with the vowel, use `'[!aeiou]*'`.

- 
- 'soccer' and 'football' match this pattern because the first character is not an 'a', 'e', 'i', 'o', or a 'u'.
 - When using character classes, you can create a range by separating two characters with a hyphen for example if you want to match all the characters from 'a' to 'g', use '[a-g]'.
 - To match the numbers - 3, 4, 5 and 6, use '[3-6]'.
 - **Predefined Classes**
 - For wildcards, we can also use predefined named character classes, and these named character classes represent the most commonly used ranges.
 - The named classes are enclosed surrounded by '['[:' on the left and ':']]' on the right.

- 
- `[[:alpha:]]` - matches alphabetic letters, both - lower and upper case letters.
 - `[[:alnum:]]` - matches alphanumeric characters, i.e. both alpha and digits. To be specific, it matches any uppercase or lowercase letters or any decimal digits.
 - `[[:digit:]]` - the numbers and decimal from 0 to 9.
 - `[[:lower:]]` - matches any lowercase letters.
 - `[[:space:]]` - matches white space, meaning characters such as spaces, tabs and newline characters.
 - `[[:upper:]]` - matches uppercase letters.

- 
- **Special Situation:** There are times when you will want to match one of the wildcard characters * or ?. In such times, you would escape that character with a back slash '\', by placing the '\' character before the wildcard character.
 - As an example, if you wanted to match all the files that end with a question mark, then use '*\?'.

I/O Redirection

- Here we will see the different types of Linux I/O (input and output) and how to redirect that I/O to the three default types of input and output, i.e. standard input, standard output and standard error.
- By default, standard input comes from the keyboard and standard output and standard error are displayed to the screen.

- 
- Since Linux represents practically everything as a file, each one of these I/O types is given a file descriptor which are just numbers that represent open files.
 - As the I/O types are represent open files, we can think of the keyboard and screen as files.

Input/Output Types

I/O Name	Abbreviation	File Descriptor
Standard Input	stdin	0
Standard Output	stdout	1
Standard Error	stderr	2



■ Making the I/O types to be treated as files gives you the ability to do some powerful things like take the standard output of one command that would normally be displayed to your screen and use it as the input to another command, or even redirect that output to a file.

Redirection

- > Redirects standard output to a file.
Overwrites (truncating) existing contents.
- >> Redirects standard output to a file.
Appends to any existing contents.
- < Redirects input from a file to a command.

- 
- We use the greater-than sign (>) to redirect output and the less-than sign (<) to redirect input.
 - The greater-than sign (>) redirects output from a command to a file and the less-than sign (<) redirects input from a file to a command.
 - The explicit way of using redirection is to provide a file descriptor number. However, if 1 is omitted, then the file descriptor 0 is assumed for input and 1 for output redirection.
 - When you use 2 greater-than (>>) signs, the redirected output will append to the file.
 - Normally with redirection of a file follows the redirection operator, so you want to use a file descriptor instead of a file name use the ampersand (&) symbol.

- 
- So instead of redirecting standard error to a file `'2>out.err'` for instance, you could redirect it to standard output with `'2>&1'`, this effectively combines standard input and standard output.

Redirection

<code>&</code>	Used with redirection to signal that a file descriptor is being used.
<code>2>&1</code>	Combine stderr and standard output.
<code>2>file</code>	Redirect standard error to a file.

- If you want to ignore output, you can send it to the null device, which is `'>/dev/null'`. The null device is a special file that throws away whatever is fed to it, so you may hear people refer to this as a 'bit bucket'.

- 
- If you prefer not to see errors on your screen and don't care to save them to a file either simply redirect them to 'dev/null'.

➤ Examples

1. Run the command 'ls -l' in your home directory and redirect that standard output (because it's displayed to the screen) to a file, using the greater-than symbol (>) and followed by a filename, use lsfiles.txt. If you 'cat' the contents of lsfiles.txt, you will notice it has the output of the 'ls -l' command.
2. You can append to the lsfiles.txt file with two greater than symbols (>>), so execute the 'ls -al' and append the standard output of 'ls -al' to 'lsfiles.txt'. Again, if you 'cat' the contents of lsfiles.txt, it has the output of the 'ls -l' command from 1 above followed by the contents of the 'ls -al' command executed in this example.

- 
- To see how 'input redirection' works, take the same file from the exercises above 'lsfiles.txt' and redirect its contents to the 'sort' command, '**sort < lsfiles.txt**'
 - Notice that 'sort' was run on the command and in the same way it would when you run '**sort lsfiles.txt**', this is because it is the same thing we are doing in different ways
 - Yet another way to do the same thing using file descriptor 1 for output redirection and file descriptor 2 for input redirection, then see that '**ls -l > files.txt**' is the same as '**ls -l 1> files.txt**', where 1 is the file descriptor number, but make sure there is no space between the file descriptor and the greater than (>) sign i.e. 1>

- 
- Input and output redirection can be combined.
 - Let's see this in action by using the 'lsfiles.txt' file as the input for 'sort' command and redirect the output to 'lssorted.txt'.
 - No issues, but let's assume there are errors. Now the errors will be displayed to the screen (standard error) while the OK results will be redirected to the 'lssorted.txt' file.
 - File descriptor 1 is for standard output, while 2 is for standard error, but file descriptor 1 is the default descriptor for all output redirection, meaning that not all the output generated by a program is captured by default.
 - Since lsfiles.txt was created above, running the command **'ls lsfiles.txt mylsfile.txt'** will find 'lsfiles.txt' but will not find mylsfile.txt

- 
- Let's run the same command, but this time we are going to redirect standard output (file descriptor 1) to the file 'outfile', and do not redirect standard error.
 - Notice that the errors were displayed on the screen because it did not go to 'outfile'.
 - Now let us redirect standard error into 'outfile.log' and leave standard output alone. Standard error went to the file and standard output went to the screen.
 - Based on the last 3 bullets, you can send standard output to one place and standard error to another place as follows
\$ ls lsfiles.txt mylsfile.txt 1> outfile 2> outfile.log
 - Standard output goes to 'outfile', while standard error goes to 'outfile.log'

- 
- You can also combine standard output and standard error sending both to the same place, a file or to /dev/null
 - To see how it works, let's redirect both standard output and standard error to 'outfile.join', i.e. we are taking standard error and appending it to standard output as shown below
\$ ls lsfiles.txt mylsfile.txt 2>&1 outfile.join
 - This means the standard error is appended to the standard input so both end up in 'outfile.join' file.

- 
- When using the null device, /dev/null, you have 3 options
 1. To send only your errors to, if you are ignoring errors.
 2. To send only your standard output, if only interested in errors
 3. To send both your standard output and standard error, if you are not interested in anything i.e. no output.

The Null Device

>/dev/null Redirect output to nowhere.

```
$ ls here not-here 2> /dev/null
here
```

```
$ ls here not-here > /dev/null 2>&1
$
```

Comparing Files

- In Linux, if you want to compare the contents of two files and display the differences, you can use the '**diff**', the '**sdiff**' or the '**vimdiff**' command.
- The '**diff**' command displays the differences between the contents of two files.
- The '**sdiff**' command will display the differences with file1 on the left and file2 on the right, while '**vimdiff**' will use the 'vim' editor to display the differences between two files.

Comparing the Contents of Files

<code>diff file1 file2</code>	Compare two files.
<code>sdiff file1 file2</code>	Side-by-side comparison.
<code>vimdiff file1 file2</code>	Highlight differences in vim.

diff Output

```
$ diff file1 file2
```

```
3c3
```

```
...
```

```
LineNumFile1-Action-LineNumFile2
```

```
Action = (A)dd (C)hange (D)elete
```

- Here above is just the first line of the output produced by the 'diff' command.
- The first number represents line numbers from the first file, the second number represents line numbers from the second file and the middle character separating the line numbers will either be a 'c' for a change, 'd' for a deletion or 'a' for an addition.

- So in this example, the 3rd line of the 1st file is changed or different from the 3rd line in the 2nd file.
- This is an example of the entire output from the 'diff' command.

diff Output

```
$ diff file1 file2
3c3
< this is a line in a file.
---
> This is a Line in a File!
                                < Line from file1
                                > Line from file2
```

- The output after the less-than sign is from the file1, first file, and the output following the greater-than sign is from file2, 2nd file, the three dashes (---) are only a separator

sdiff Output

```
$ sdiff file1 file2
line in file1      |    line in file2
                   >    more in file2
                   | Differing lines
                   < Line from file1
                   > Line from file2
```

- Following an 'sdiff' command, the pipe or vertical bar character (|) in the output is an indication that the text differs in the files on that line.
- In addition, if you see a less-than (<) sign, it means that line only exists in the 1st file and a greater-than (>) sign means that line only exists in the 2nd file.

vimdiff

<code>Ctrl-w w</code>	Go to next window
<code>:q</code>	Quit (close current window)
<code>:qa</code>	Quit all (close both files)
<code>:qa!</code>	Force quit all

- Running 'vimdiff', both files will be pulled up in separate windows.
- To switch between the windows, type 'Ctrl-ww'.
- To close the file in the current window, type ':q', enter.
- To close both files at once and quit, type ':qa'.
- Finally, if you've made any changes that you don't want to save, type ':qa!', enter.



➤ Practice:

- We'll cat 2 files compare and compare1 with the '-n' option so the line numbers are displayed, then use the 'diff' command to see the differences between the two files.
- The line that begins with a less-than (<) symbol, is from the 1st file and the line with the greater-than (>) symbol is from the 2nd file.
- You also know the 1st line of 'diff' output says '2c2', meaning that the 2nd line of the 1st file is different from the 2nd line of the 2nd file.
- If we use 'sdiff', notice that the files are side by side and the vertical bar or the pipe symbol (|) indicates a line with differences.

- 
- To have 'sdiff' display the < and > symbols, append a line to compare and a different line to compare1.
 - Now run 'sdiff' again, you will see a less than (<) for the new line that was added in file 'compare' and a greater than (>) for the new line that was added in file 'compare1'
 - Using 'vimdiff', 'vim' will highlight the differences between the files.
 - 'Ctrl-ww' will move you from one window to the other window.
 - ':q' will close the current window, and another ':q' will close the other window. ':qa', will close both windows at the same time and exit 'vimdiff'.
 - ':qa!', will ignore any changes, close both windows and exit 'vimdiff'.

Searching in Files and Using Pipes

- Searching the contents of files, especially log files, is a very important skill for Systems Admins, so that is what we will be looking at in the lesson.
- We will also look at pipes and what pipes are and also how to use them.
- You use the '**grep**' command to search for text within a file and the simplest format for the '**grep**' command is **\$ grep pattern < search file >**, which indicates you are searching for pattern in <search file>.

The grep Command

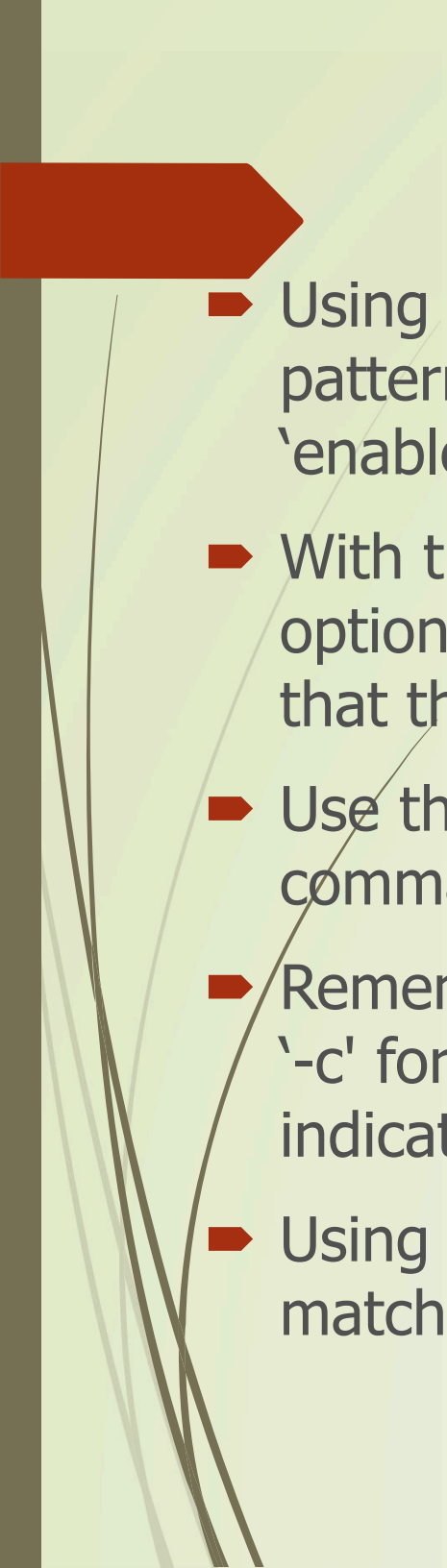
grep Display lines matching a pattern.

grep pattern file

grep Options

- i Perform a search, ignoring case.
- c Count the number of occurrences in a file.
- n Precede output with line numbers.
- v Invert Match. Print lines that don't match.

- As listed above, here are some options that you can use with the **grep** command.
- '-i' performs a case insensitive search.
- '-c' displays the number of occurrences found in the file.
- '-n' precedes the output with line numbers.
- '-v' does an invert match, that is it returns only the files that do not contain pattern

- 
- Using the '**grep**' command, search the 'grepfile' file for the pattern 'enable'. '**grep**' will find all the occurrences of 'enable' found in the file i.e. matched the 'enable' pattern
 - With the same pattern, let's invert the search using the '-v' option and all the lines with enable will be excluded, notice that the search is case sensitive.
 - Use the '-i' option to ignore case and now the '**grep**' command will return the lines with enable in any case.
 - Remember that you can also combine options, so we will do '-c' for count and '-I' for ignore case and '**grep**' will indicate the number of lines that match enable.
 - Using the '-n' option will display which line number that the match occurs on.

- 
- Just looking at the names of files can give an indication what type of data the file may contain.
 - If a file ends in '.txt', chances are it's a text file. If the file has executable permission set, it might be an executable program.
 - That is primarily guess work, but there is a way to determine the content type of a file.
 - We determine the type of a file's content by running the '**file**' command. If it's a text file, it will say that it's text in the output of the file command.
 - A binary file is a file that is in machine readable format and not a human readable format.

- 
- If you run '**grep**' against a binary file, it simply displays whether or not that information was found in the file but it will not display the text.
 - To look at the textual data within a binary file, use the '**strings**' command.

The file Command

```
file file_name Display the file type.  
$ file sales.data  
sales.data: ASCII text  
$ file *  
bin: directory  
jason.tar: POSIX tar archive
```

Searching for Text in Binary Files

`strings` Display printable strings.

➤ Pipes

- In Linux, the vertical bar is called a pipe, and you can use it to join commands together.

Pipes

| Pipe symbol

`command-output | command-input`

- As you see above, pipes takes the standard output from the preceding command and passes it as the standard input to the next command.

- 
- If errors are generated by the first command, those error messages, by default, are not passed to the second command and that is because error messages are displayed on standard error.
 - To pass standard error as input to the next command after the pipe, you can redirect standard error to standard input.

Pipes

```
grep pattern file  
cat file | grep pattern
```

- Above is a common pattern you will see often in Linux and there are several commands that allow you to specify a file on the command line that is to be used for input.

- 
- With 'grep' for example, you can run 'grep', followed by the pattern, followed by the file to use as input and if you don't supply a file name the command will use standard input.
 - 'grep' pattern file is equivalent to 'cat file | grep pattern'.
 - That is because 'cat' displays the contents of the file on standard output, the pipe takes the standard output from the 'cat' command and passes it in as a standard input to the 'grep' command, since no file name was specified for grep to act on, it takes standard input as its input

- 
- The '**cut**' command allows you to select portions of a file or to cut out pieces of a file and like grep, if file is omitted, cut will use standard input.

The cut Command

<code>cut [file]</code>	Cut out selected portions of file. If file is omitted, use standard input.
-------------------------	--

cut Options

<code>-d delimiter</code>	Use delimiter as the field separator.
<code>-f N</code>	Display the Nth field.

- 
- The '-d' option lets you to specify a field delimiter, '-f' tells cut, which field to display.
 - Examples!!
 - Let's search for 'enable' in the grepfile using '**grep -n**', we see that grep tells us more than one piece of information.
 - Now use the '**cut**' command to cut the first field of the output from the 'grep'command
 - `$ grep -n enable grepfile | cut -d: -f1`
 - Pipes are not limited to two commands because you can chain as many commands together as you want, example
 - `$ grep -n enable grepfile | cut -d: -f1 | head -1`



Searching and Pipe Example

- Find all users with home directories under "home" in /etc/passwd.
- Print account name and real name.
- Print in alphabetical order by account name.
- Print in a tabular format.

- ➡ Per the image above, we want to find all regular users who have a home directory under /home.
- ➡ The /etc/passwd file contains a list of all the accounts on the Linux system.
- ➡ From that file, we want to print each account name and their real name, nothing else.
- ➡ We want to print them in alphabetical order by account name and we want to display them in a tabular format.

- 
- Let's look for users with home directories under /home in etc/passwd.

```
$ grep user /etc/passwd or cat /etc/passwd | grep user
```

- There is a lot of information in the /etc/passwd file but we want to print the account name and real name only

We use the cut command to get the information we want. The fields that are separated by a colon, so we can use '-d' and the colon as a field delimiter. Account name is the first field and the fifth field contains the real name, so use '-f1,5'.

```
$ cat /etc/passwd | grep user | cut -d: -f1,5 or
```

```
$ grep user /etc/passwd | cut -d: -f1,5
```

- I also want to sort them in alphabetical order.

```
$ grep user /etc/passwd | cut -d: -f1,5 | sort
```

- 
- We want to get rid of the colon character.

We use a command called, '**tr**' that translates characters. We'll use it to translate the colon character into a space character.

```
$ grep user /etc/passwd | cut -d: -f1,5 | sort | tr ':' ' '
```

- Finally we want to display this information in a table format we use the '**column**' command and the '-t' for table to achieve this

```
$ grep user /etc/passwd | cut -d: -f1,5 | sort | tr ':' ' ' | column -t
```

Piping Output to a Pager

- more
- less

- 
- Another common use of pipes is to control how output is displayed to your screen.

Piping Output to a Pager

- more
- less

- This becomes handy if a command produces a significant amount of output that can scroll off your screen before you have a chance to view it.
- To control the output, use a pager utility such as more or less. `$ cat /etc/passwd | grep ou | [ more | less ]`
- We've already used these commands already on files directly, but keep in mind that they can take a redirected input as well.

