IBM

developerWorks®

Learn Linux, 101: Search text files using regular expressions

Finding needles in haystacks

Ian Shields
Senior Programmer
IBM

18 February 2013
(First published 03 February 2010)

Learn how to use regular expressions, and then use them to find things in files on your filesystem. You can use the material in this article to study for the LPI 101 exam for Linux® system administrator certification, or just to learn for fun.
18 Feb 2013 - *In response to reader comment, added new first line of code to Listing 11.*

View more content in this series

Overview

This article grounds you in the basic Linux techniques for searching text files using regular expressions. Learn to:

- Create simple regular expressions
- Search files and filesystems using regular expressions
- Use regular expressions with sed

This article helps you prepare for Objective 103.7 in Topic 103 of the Linux Professional Institute's Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 2.

Prerequisites

To get the most from the articles in this series, you should have a basic knowledge of Linux and a working Linux system on which you can practice the commands covered in this article. Sometimes different versions of a program will format output differently, so your results may not always look exactly like the listings and figures shown here. This article builds on the concepts discussed in the earlier article "Learn Linux 101: Text streams and filters."

About this series

This series of articles helps you learn Linux system administration tasks. You can also use the material in these articles to prepare for Linux Professional Institute Certification level 1 (LPIC-1) exams.

See our series roadmap for a description of and link to each article in this series. The roadmap is in progress and reflects the latest (April 2009) objectives for the LPIC-1 exams: as we complete articles, we add them to the roadmap. In the meantime, though, you can find earlier versions of similar material, supporting previous LPIC-1 objectives prior to April 2009, in our LPI certification exam prep tutorials.

Setting up the examples

Connect with Ian

Ian is one of our most popular and prolific authors. Browse all of Ian's articles on developerWorks. Check out Ian's profile and connect with him, other authors, and fellow readers in My developerWorks.

In this article, we will practice the commands using some of the files created in the article "Learn Linux 101: Text streams and filters." In case you did not already do that or did not save the files you worked with, you can start by creating a new subdirectory in your home directory called lpi103-7 and creating the necessary files in it. Do so by opening a text window with your home directory as your current directory. Then copy the contents of Listing 1 into the window to run the commands that will create the lpi103-7 subdirectory and the files you will use.

Listing 1. Creating the example files

```
mkdir -p lpi103-7 && cd lpi103-7 && {
echo -e "1 apple\n2 pear\n3 banana" > text1
echo -e "9\tplum\n3\tbanana\n10\tapple" > text2
echo "This is a sentence. " !#:* !#:1->text3
split -l 2 text1
split -b 17 text2 y;
cp text1 text1.bkp
mkdir -p backup
cp text1 backup/text1.bkp.2
}
```

You window should look similar to Listing 2, and your current directory should now be the newly created lpi103-7 directory.

Listing 2. Creating the example files -- output

```
ian@attic4:~$ mkdir -p lpi103-7 && cd lpi103-7 && {
> echo -e "1 apple\n2 pear\n3 banana" > text1
> echo -e "9\tplum\n3\tbanana\n10\tapple" > text2
> echo "This is a sentence. " !#:* !#:1->text3
echo "This is a sentence. " "This is a sentence. " "This is a sentence. ">text3
> split -l 2 text1
> split -b 17 text2 y;
> cp text1 text1.bkp
> mkdir -p backup
> cp text1 backup/text1.bkp.2
> }
ian@attic4:~/lpi103-7$
```

Regular expressions

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Basics of Linux system administration: Working at the console

Regular expressions have their roots in computer language theory. Most students of computer science learn that the languages that can be denoted by regular expressions are precisely the same as those accepted by finite automata. The regular expressions covered in this article are capable of expressing more complexity and so are **not** the same as those you may have learned about in computer science classes, although the heritage is clear.

A *regular expression* (also called a "regex" or "regexp") is a way of describing a text string or *pattern* so that a program can *match* the pattern against arbitrary text strings, providing an extremely powerful search capability. The `grep` (for *g*eneralized *r*egular *e*xpression *p*rocessor) is a standard part of any Linux or UNIX® programmer's or administrator's toolbox, allowing regular expressions to be used in file searches or command output. In the article "Learn Linux 101: Text streams and filters," we introduced `sed`, the *s*tream *ed*itor, which is another standard tool that uses regular expressions extensively for finding and replacing text in files or text streams. This article helps you better understand the regular expressions used by both `grep` and `sed`. Another program that uses regular expression extensively is `awk`.

As with other parts of this article series, whole books have been written on regular expressions and computer language theory. See Resources for some suggestions.

As you learn about regular expressions, you will see similarities between regular expression syntax and the wildcard (or globbing) syntax discussed in the article "Learn Linux 101: File and directory management." The similarity is only superficial.

Basic building blocks

Two forms of regular expression syntax are used with the GNU grep program found on most Linux systems: *basic* and *extended*. With GNU grep, there is no difference in functionality. Basic syntax is described here, along with the differences between it and extended syntax.

Regular expressions are built from *characters* and *operators*, augmented by *metacharacters*. Most characters match themselves, and most metacharacters must be escaped using a backslash (\). The fundamental operations are:

Concatenation
Concatenating two regular expressions creates a longer expression. For example, the regular expression **a** will match the string **abcdcba** twice (the first and last **a**) and so will the regular expression **b**. However, **ab** will match only **ab**cdcba, while **ba** will match only abcdc**ba**.

Repetition
The Kleene * or repetition operator will match zero or more occurrences of the preceding regular expression. Thus an expression like **a*b** will match any string of **a**s terminated by a **b**, including just **b** itself. The Kleene * does not have to be escaped, so an expression in which you want to match a literal asterisk (*) must have the asterisk escaped. The use of * here is different from the use in globbing, where it matches any string.

Alternation
The alternation operator (|) matches either the preceding or following expression. It must be escaped in basic syntax. For example, the expression **a*\|b*c** will match a string consisting

of any number of **a**s or any number of **b**s (but not both) terminated by a single **c**. Again, the single character **c** matches.

You often need to quote your regular expressions to avoid shell expansion.

Search files and filesystems

We will use the text files that we created earlier as examples (see "Setting up the examples"). Study the simple examples of Listing 3. Note that `grep` takes a regular expression as a required parameter and a list of zero or more files to search. If no files are given, grep searches stdin, which makes it a filter that can be used in pipelines. If no lines match, there is no output from `grep`, although its exit code can be tested.

Listing 3. Simple regular expressions

```
ian@attic4:~/lpi103-7$ grep p text1
1 apple
2 pear
ian@attic4:~/lpi103-7$ grep pea text1
2 pear
ian@attic4:~/lpi103-7$ grep "p*" text1
1 apple
2 pear
3 banana
ian@attic4:~/lpi103-7$ grep "pp*" text1
1 apple
2 pear
ian@attic4:~/lpi103-7$ grep "x" text1; echo $?
1
ian@attic4:~/lpi103-7$ grep "x*" text1; echo $?
1 apple
2 pear
3 banana
0
ian@attic4:~/lpi103-7$ cat text1 | grep "l\|n"
1 apple
3 banana
ian@attic4:~/lpi103-7$ echo -e "find an \ns* here" | grep "s\*"
s* here
```

As you can see from these examples, you may sometimes get surprising results, particularly with repetition. You may have expected **p*** or at least **pp*** to match a couple of **p**s, but **p***, and **x*** too, match every line in the file because the * operator matches **zero** or more of the preceding regular expression.

Two of the examples illustrate the exit code from grep. A value of 0 is returned if a match is found, and a value of 1 if no match is found. A value greater than 1 (always 2 for GNU grep) is returned in the event of an error, such as the file you are attempting to search not existing.

First shortcuts

Now that you can use the basic building blocks of regular expressions with `grep`, here are some convenient shortcuts.

+
: The + operator is like the * operator, except that it matches **one** or more occurrences of the preceding regular expression. It must be escaped for basic expressions.

?
: The ? indicates that the preceding expression is optional, so it represents zero or one occurrences of it. This is not the same as the ? used in globbing.

.
: The . (dot) is a metacharacter that stands for any character. One of the most commonly used patterns is **.***, which matches an arbitrary length string containing any characters (or no characters at all). Needless to say, you will find this used as part of a longer expression. Compare a single dot with the ? used in globbing and .* with the * used in globbing.

Listing 4. More regular expressions

```
ian@attic4:~/lpi103-7$ grep "pp\+" text1 # at least two p's
1 apple
ian@attic4:~/lpi103-7$ grep "pl\?e" text1
1 apple
2 pear
ian@attic4:~/lpi103-7$ grep "pl\?e" text1 # pe with optional l between
1 apple
2 pear
ian@attic4:~/lpi103-7$ grep "p.*r" text1 # p, some string then r
2 pear
ian@attic4:~/lpi103-7$ grep "a.." text1 # a followed by two other letters
1 apple
3 banana
```

Matching beginning or end of line

The ^ (caret) matches the beginning of a line while the $ (dollar sign) matches the end of line. So **^..b** matches any two characters at the beginning of a line followed by a **b**, while **ar$** matches any line ending in **ar**. The regular expression **^$** matches an empty line.

More complex expressions

So far, we have seen repetition applied to a single character. If you wanted to search for one or more occurrences of a multicharacter string such as the **an** that occurs twice in b**anan**a, use parentheses, which must be escaped in basic syntax. Similarly, you might want to search for a few characters, without using something as general as the . or as long-winded as alternation. You can do this too, by enclosing the alternatives in square brackets ([]), which do not need to be escaped for regular syntax. Expressions in square brackets constitute a *character class*. With a few exceptions covered later, using square brackets also eliminates the need for escaping special characters such as . and *.

Listing 5. Parentheses and character classes

```
ian@attic4:~/lpi103-7$ grep "\(an\)\+" text1 # find at least 1 an
3 banana
ian@attic4:~/lpi103-7$ grep "an\(an\)\+" text1 # find at least 2 an's
3 banana
ian@attic4:~/lpi103-7$ grep "[3p]" text1 # find p or 3
1 apple
2 pear
3 banana
ian@attic4:~/lpi103-7$ echo -e "find an\ns* here\nsomewhere." | grep "s[.*]"
s* here
ian@attic4:~/lpi103-7$ echo -e "find an\n * in position 2." | grep ".[.*]"
 * in position 2.
```

There are several other interesting possibilities for character classes.

Range expression
A range expression is two characters separated by a - (hyphen), such as 0-9 for digits, or 0-9a-fA-F for hexadecimal digits. Note that ranges are locale-dependent.

Named classes
Several named classes provide a convenient shorthand for commonly used classes. Named classes open with [: and close with :] and may be used within bracket expressions. Some examples:

[:alnum:]
Alphanumeric characters

[:blank:]
Space and tab characters

[:digit:]
The digits 0 through 9 (equivalent to 0-9)

[:upper:] and [:lower:]
Upper and lower case letters, respectively.

^ (negation)
When used as the first character after [in a character class, the ^ (caret) negates the sense of the remaining characters so the match occurs only if a character (except the leading ^) is **not in the class.**

Given the special meanings above, if you want to match a literal - (hyphen) within a character class, you must put it first or last. If you want to match a literal ^ (caret), it must not be first. And a] (right square bracket) closes the class, unless it is placed first.

Character classes are one area where regular expressions and globbing **are** similar, although the negation differs (^ vs. !). Listing 6 shows some examples of character classes.

Listing 6. More character classes

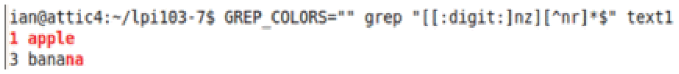
```
ian@attic4:~/lpi103-7$ # Match on range 3 through 7
ian@attic4:~/lpi103-7$ echo -e "123\n456\n789\n0" | grep "[3-7]"
123
456
789
ian@attic4:~/lpi103-7$ # Find digit followed by no n or r till end of line
ian@attic4:~/lpi103-7$ grep "[[:digit:]][^nr]*$" text1
1 apple
ian@attic4:~/lpi103-7$ # Find a digit, n, or z followed by no n or r till end of line
ian@attic4:~/lpi103-7$ grep "[[:digit:]nz][^nr]*$" text1
1 apple
3 banana
```

Does the last example surprise you? In this case, the first bracket expression matches **any** digit, n or z in the string and the last n is not followed by another n or r, so the trailing na of the string matches the regular expression.

What was matched?

If you able to distinguish highlighting, such as color, bold, or underscore, you can set the GREP_COLORS environment variable to highlight matches. The default setting displays matches highlighted in bold red as illustrated in Figure 1. You see that the whole first line of output matches, but only the last two characters of the second line match.

Figure 1. Using color for grep matches



```
ian@attic4:~/lpi103-7$ GREP_COLORS="" grep "[[:digit:]nz][^nr]*$" text1
1 apple
3 banana
```

If you're new to regular expressions, or unsure as to why a particular line was returned by grep, this technique may help you.

Extended regular expressions

Extended regular expression syntax is a GNU extension. It eliminates the need to escape several characters when used as we have used them in basic syntax, including parentheses, '?', '+', '|', and '{'. The flip side is that you must escape them if you want them interpreted as characters in your regular expression. You may use the `-E` (or `--extended-regexp` option of grep) to indicate that you are using extended regular expression syntax. Alternatively, the `egrep` command does this for you. Listing 7 shows an example used earlier in this section along with the corresponding extended expression used with `egrep`.

Listing 7. Extended regular expressions

```
ian@attic4:~/lpi103-7$ # Find b followed by one or more an's and then an a
ian@attic4:~/lpi103-7$ grep "b\(an\)\+a" text1
3 banana
ian@attic4:~/lpi103-7$ egrep "b(an)+a" text1
3 banana
```

Finding stuff in files

Now that you have command of the basics, let's use the power of `grep` and `find` to hunt down things in your filesystem. Again, the examples are relatively simple; they use the files created in an earlier article or the ones you created in the lpi103-7 directory and its children. (See "Setting up the examples.") If you use files from the earlier article in this series, you will have some extra files and you will see some extra results.

First, `grep` can search multiple files at once. If you add the `-n` option, it tells you what line numbers matched. If you simply want to know how many lines matched, use the `-c` option, and if you want just a list of files with matches, use the `-l` option. Listing 8 shows some examples.

Listing 8. Grepping multiple files

```
ian@attic4:~/lpi103-7$ grep plum *
text2:9 plum
yaa:9 plum
ian@attic4:~/lpi103-7$ grep -n banana text[1-4]
text1:3:3 banana
text2:2:3 banana
ian@attic4:~/lpi103-7$ grep -c banana text[1-4]
text1:1
text2:1
text3:0
ian@attic4:~/lpi103-7$ grep -l pear *
text1
text1.bkp
xaa
```

If you look at the use of the `-c` option in Listing 8, you will see a line `text3:0`. You will frequently want to know how many occurrences of something are in a file, but you won't want to know the files that don't have any occurrences of whatever you are looking for. The `grep` command has a `-v` option, which tells it to only display output for lines that do **not** match. So we could use the regular expression `:0$` to find lines that end with a colon and 0.

Our next example does this by using `find` to locate all the regular files in the current directory and its children, and then uses `xargs` to pass the file list to `grep` to determine the number of occurrences of **banana** in each file. Finally, this output is filtered through another invocation of `grep`, this time with the `-v` option to find all lines that do **not** end with :0, leaving us only with counts for files that actually contain the string **banana**.

Listing 9. Hunting down files with at least one banana

```
ian@attic4:~/lpi103-7$ find . -type f -print0| xargs -0 grep -c banana| grep -v ":0$"
./backup/text1.bkp.2:1
./text2:1
./text1:1
./yaa:1
./xab:1
./text1.bkp:1
```

Regular expressions and sed

The introduction to sed, the Stream Editor, in the article "Learn Linux 101: Text streams and filters," mentioned that sed uses regular expressions. Regexps can be used in address expressions as well as in substitution expressions.

If you're just looking for something, then you'd probably just use `grep`. If you need to extract the search string, or a related string, from the lines that match, and then manipulate it further, you might choose `sed` instead. So let's explore how this works. Recall first that our two example files, text1 and text2, contain a number followed by white space and then a fruit name, while text3 contains a repeated sentence. We show the contents again in Listing 10.

Listing 10. Contents of text1, text2 and text3

```
ian@attic4:~/lpi103-7$ cat text[1-3]
1 apple
2 pear
3 banana
9 plum
3 banana
10 apple
This is a sentence.  This is a sentence.  This is a sentence.
```

First, we'll use both `grep` and `sed` to extract just the lines that start with one or more digits followed by blank characters (space or tab). Normally, `sed` prints out every line at the end of a cycle, so we'll use the `-n` option of sed to suppress output, then use the `p` command within `sed` to print only lines that match our regular expression. To confirm that we use the same regular expression for both tools, we'll assign it to a variable.

Listing 11. Searching using both grep and sed

```
ian@attic4:~/lpi103-7$ oursearch='^[[:digit:]][[:digit:]]*[[:blank:]]'
ian@attic4:~/lpi103-7$ grep "$oursearch" text[1-3]
text1:1 apple
text1:2 pear
text1:3 banana
text2:9 plum
text2:3 banana
text2:10 apple
ian@attic4:~/lpi103-7$ cat text[1-3] | sed -ne "/$oursearch/p"
1 apple
2 pear
3 banana
9 plum
3 banana
10 apple
```

Note that `grep` will display the file name whenever more than one file is searched. Because we used `cat` to supply the input for `sed`, there is no way for `sed` to know the original file names. However, the matching lines are identical, as we would expect.

Now suppose we want only the second word of the lines that we find. This would be the name of the fruit in this case, but we might be looking for HTTP URLs, or file names or almost anything else. For our examples, it suffices to remove exactly the string that we were trying to match, so let's do it as shown in Listing 12.

Listing 12. Removing the leading numbers with sed

```
ian@attic4:~/lpi103-7$ cat text[1-3] | sed -ne "/$oursearch/s/$oursearch//p"
apple
pear
banana
plum
banana
apple
```

For our final example, suppose that our lines might have something after the fruit name. We'll add a line that says "lemon pie" to the mix and see how to just get the lemon out. We'll also sort the output and drop non-unique values, so we get a list of the fruits that are found, with each fruit appearing only once.

Listing 13 shows two ways of accomplishing the same task. In the first, we first strip off the leading number and white space that follows it, then we strip off anything after the first blank or tab and print what remains. In the second example, we introduce parentheses to break the whole line into three parts, the number and following white space, the second word, and anything else. We use the `s` command to replace the whole line with only the second word and then print the result. You might want to try a variant of this by omitting the third part, \(.*\), and see if you can explain what happens.

Listing 13. Getting to the core of the fruit

```
ian@attic4:~/lpi103-7$ echo "7 lemon pie" | cat - text[1-3] |
> sed -ne "/$oursearch/s/\($oursearch\)\([^[:blank:]]*\)\(.*\)/\2/p" |
> sort | uniq
apple
banana
lemon
pear
```

Some older versions of `sed` do not support extended regular expressions. If your version of `sed` does support extended regexps, use the `-r` option to tell `sed` that you are using extended syntax. Listing 14 shows the changes needed to the `oursearch` variable and the changes needed to the `sed` command to do the same with extended regular expressions as was done in Listing 13 with basic regular expressions.

Listing 14. Using extended regular expressions with sed

```
ian@attic4:~/lpi103-7$ echo "7 lemon pie" | cat - text[1-3] |
> sed -nre "/$oursearchx/s/($oursearchx)([^[:blank:]]*)(.*)/\2/p" |
> sort | uniq
apple
banana
lemon
pear
plum
```

This article has only scratched the surface of what you can do with the Linux command line using regular expressions with `grep` and `sed`. Use the man pages to learn more about these invaluable tools.

Resources

Learn

- Develop and deploy your next app on the IBM Bluemix cloud platform.
- Use the developerWorks roadmap for LPIC-1 to find the developerWorks articles to help you study for LPIC-1 certification based on the April 2009 objectives.
- At the LPIC Program site, find detailed objectives, task lists, and sample questions for the three levels of the Linux Professional Institute's Linux system administration certification. In particular, see their April 2009 objectives for LPI exam 101 and LPI exam 102. Always refer to the LPIC Program site for the latest objectives.
- Review the entire LPI exam prep series on developerWorks to learn Linux fundamentals and prepare for system administrator certification based on earlier LPI exam objectives prior to April 2009.
- In "Basic tasks for new Linux developers" (developerWorks, March 2005), learn how to open a terminal window or shell prompt and much more.
- The Linux Documentation Project has a variety of useful documents, especially its HOWTOs.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tutorials and Linux tips on developerWorks.
- Stay current with developerWorks technical events and Webcasts.
- Follow developerWorks on Twitter.

Get products and technologies

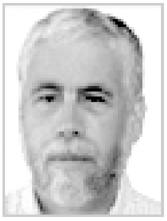
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Participate in the discussion forum for this content.
- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Ian Shields



Ian Shields works on a multitude of Linux projects for the developerWorks Linux zone. He is a Senior Programmer at IBM at the Research Triangle Park, NC. He joined IBM in Canberra, Australia, as a Systems Engineer in 1973, and has since worked on communications systems and pervasive computing in Montreal, Canada, and RTP, NC. He has several patents and has published several papers. His undergraduate degree is in pure mathematics and philosophy from the Australian National University. He has an M.S. and Ph.D. in computer science from North Carolina State University. Learn more about Ian in Ian's profile on developerWorks Community.