Learn Linux, 101: Text streams and filters

Manipulating text at the command line using GNU textutils

Ian Shields
Senior Programmer
IBM

26 January 2010
(First published 26 August 2009)

There's a lot more to text manipulation than cut and paste, particularly when you aren't using a GUI. Study for the Linux Professional Institute Certification (LPIC) 101 exam, or learn for fun. In this article, Ian Shields introduces you to text manipulation on Linux® using filters from the GNU textutils package. By the end of this article, you will be manipulating text like an expert. [*The first line of Listing 7 has been corrected, thanks to an alert reader. -Ed.*]

View more content in this series

Overview

This article gives you an introduction to *filters*, which allow you to build complex *pipelines* to manipulate text using the filters. You will learn how to display text, sort it, count words and lines, and translate characters, among other tasks. You will also learn how to use the stream editor *sed*.

In this article, you learn about the following topics:

- Sending text files and output streams through text utility filters to modify the output
- Using standard UNIX commands found in the GNU textutils package
- Using the editor sed to script complex changes to text files

This article helps you prepare for Objective 103.2 in Topic 103 of the Junior Level Administration (LPIC-1) exam 101. The objective has a weight of 3. The material in this article corresponds to the April 2009 objectives for exam 101. You should always refer to the objectives for the definitive requirements.

Text filtering

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

Text filtering is the process of taking an input stream of text and performing some conversion on the text before sending it to an output stream. Although either the input or the output can come

from a file, in the Linux and UNIX® environments, filtering is most often done by constructing a pipeline of commands where the output from one command is *piped* or *redirected* to be used as input to the next. Pipes and redirection are covered more fully in the article on *streams, pipes, and redirects* (which you can find in the series roadmap), but for now, let's look at pipes and basic output redirection using the | and > operators.

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.

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.

Streams

A *stream* is nothing more than a sequence of bytes that can be read or written using library functions that hide the details of an underlying device from the application. The same program can read from, or write to a terminal, file, or network socket in a device-independent way using streams. Modern programming environments and shells use three standard I/O streams:

- *stdin* is the *standard input stream*, which provides input to commands.
- *stdout* is the *standard output stream*, which displays output from commands.
- *stderr* is the *standard error stream*, which displays error output from commands.

Piping with |

Input can come from parameters you supply to commands, and output can be displayed on your terminal. Many text processing commands (filters) can take input either from the standard input stream or from a file. In order to use the output of a command, command1, as input to a filter, command2, you connect the commands using the pipe operator (|). Listing 1 shows how to pipe the output of `echo` to sort a small list of words.

Listing 1. Piping output from echo to input of sort

```
[ian@echidna ~]$ echo -e "apple\npear\nbanana"|sort
apple
banana
pear
```

Either command may have options or arguments. You can also use | to redirect the output of the second command in the pipeline to a third command, and so on. Constructing long pipelines of commands that each have limited capability is a common Linux and UNIX way of accomplishing

tasks. You will also sometimes see a hyphen (-) used in place of a filename as an argument to a command, meaning the input should come from stdin rather than a file.

Output redirection with >

While it is nice to be able to create a pipeline of several commands and see the output on your terminal, there are times when you want to save the output in a file. You do this with the output redirection operator (>).

For the rest of this section, we will be using some small files, so let's create a directory called lpi103-2 and then cd into that directory. We will then use > to redirect the output of the `echo` command into a file called text1. This is all shown in Listing 2. Notice that the output does not display on the terminal because it has been redirected to the file.

Listing 2. Redirecting output from a command to a file

```
[ian@echidna ~]$ mkdir lpi103-2
[ian@echidna ~]$ cd lpi103-2
[ian@echidna lpi103-2]$ echo -e "1 apple\n2 pear\n3 banana" > text1
```

Now that we have a couple of basic tools for pipelining and redirection, let's look at some of the common UNIX and Linux text processing commands and filters. This section shows you some of the basic capabilities; check the appropriate man pages to find out more about these commands.

Cat, od, and split

Now that you have created the test1 file, you might want to check what is in it. Use the `cat` (short for *concatenate)* command to display the contents of a file on stdout. Listing 3 verifies the contents of the file created above.

Listing 3. Displaying file contents with cat

```
[ian@echidna lpi103-2]$ cat text1
1 apple
2 pear
3 banana
```

The `cat` command takes input from stdin if you do not specify a filename (or if you specify - as the filename). Let's use this along with output redirection to create another text file as shown in Listing 4.

Listing 4. Creating a text file with cat

```
[ian@echidna lpi103-2]$ cat >text2
9       plum
3       banana
10      apple
```

Many little filters

Another example of a small filter is the `tac` command. The name is the reverse of `cat`, and the function is also the reverse of `cat`, in that the file is displayed in reverse order. Try running

```
tac text2 text1
```
for yourself.

In Listing 4, `cat` keeps reading from stdin until end of file. Use the **Ctrl-d** (hold **Ctrl** and press **d**) combination to signal end of file. This is the same key combination to exit from the bash shell. Use the tab key to line up the fruit names in a column.

Remember that *cat* was short for *concatenate*? You can use `cat` to concatenate several files together for display. Listing 5 shows the two files we have just created.

Listing 5. Concatenating two files with cat

```
[ian@echidna lpi103-2]$ cat text*
1 apple
2 pear
3 banana
9       plum
3       banana
10      apple
```

When you display these two text files using `cat`, you notice alignment differences. To learn what causes this, you need to look at the control characters that are in the file. These are acted on in text display output rather than having some representation of the control character itself displayed, so we need to *dump* the file in a format that allows you to find and interpret these special characters. The GNU text utilities include an `od` (or *O*ctal *D*ump) command for this purpose.

There are several options for `od`, such as the `-A` option to control the radix of the file offsets and `-t` to control the form of the displayed file contents. The radix may be specified as o, (octal, the default), d (decimal), x (hexadecimal), or n (no offsets displayed). You can display output as octal, hex, decimal, floating point, ASCII with backslash escapes, or named characters (nl for newline, ht for horizontal tab, etc.). Listing 6 shows some of the formats available for dumping the text2 example file.

Listing 6. Dumping files with od

```
[ian@echidna lpi103-2]$ od text2
0000000 004471 066160 066565 031412 061011 067141 067141 005141
0000020 030061 060411 070160 062554 000012
0000031
[ian@echidna lpi103-2]$ od -A d -t c text2
0000000   9  \t   p   l   u   m  \n   3  \t   b   a   n   a   n   a  \n
0000016   1   0  \t   a   p   p   l   e  \n
0000025
[ian@echidna lpi103-2]$ od -A n -t a text2
   9  ht   p   l   u   m  nl   3  ht   b   a   n   a   n   a  nl
   1   0  ht   a   p   p   l   e  nl
```

Notes:

1. The `-A` option of `cat` provides an alternate way of seeing where your tabs and line endings are. See the man page for more information.
2. If you see spaces instead of tabs in your own text2 file, refer to Expand, unexpand, and tr, later in this article to see how to switch between tabs and spaces in a file.

3. If you have a mainframe background, you may be interested in the hexdump utility, which is part of a different utility set. It's not covered here, so check the man pages.

Our sample files are very small, but sometimes you will have large files that you need to split into smaller pieces. For example, you might want to break a large file into CD-sized chunks so you can write it to CD for sending through the mail to someone who could create a DVD for you. The `split` command will do this in such a way that the `cat` command can be used to recreate the file easily. By default, the files resulting from the `split` command have a prefix in their name of 'x' followed by a suffix of 'aa', 'ab', 'ac', ..., 'ba', 'bb', etc. Options permit you to change these defaults. You can also control the size of the output files and whether the resulting files contain whole lines or just byte counts.

Listing 7 illustrates splitting our two text files with different prefixes for the output files. We split text1 into files containing at most two lines, and text2 into files containing at most 18 bytes. We then use `cat` to display some of the pieces individually as well as to display a complete file using *globbing*, which is covered in the article on *basic file and directory management* (which you can find in the series roadmap).

Listing 7. Splitting and recombining with split and cat

```
[ian@echidna lpi103-2]$ split -l 2 text1
[ian@echidna lpi103-2]$ split -b 17 text2 y
[ian@echidna lpi103-2]$ cat yaa
9       plum
3       banana
1[ian@echidna lpi103-2]$ cat yab
0       apple
[ian@echidna lpi103-2]$ cat y* x*
9       plum
3       banana
10      apple
1 apple
2 pear
3 banana
```

Note that the split file named yaa did not finish with a newline character, so our prompt was offset after we used `cat` to display it.

Wc, head, and tail

`Cat` displays the whole file. That's fine for small files like our examples, but suppose you have a large file. Well, first you might want to use the `wc` (*W*ord *C*ount) command to see how big the file is. The `wc` command displays the number of lines, words, and bytes in a file. You can also find the number of bytes by using `ls -l`. Listing 8 shows the long format directory listing for our two text files, as well as the output from `wc`.

Listing 8. Using wc with text files

```
[ian@echidna lpi103-2]$ ls -l text*
-rw-rw-r--. 1 ian ian 24 2009-08-11 14:02 text1
-rw-rw-r--. 1 ian ian 25 2009-08-11 14:27 text2
[ian@echidna lpi103-2]$ wc text*
 3  6 24 text1
 3  6 25 text2
 6 12 49 total
```

Options allow you to control the output from `wc` or to display other information such as maximum line length. See the man page for details.

Two commands allow you to display either the first part (*head*) or last part (*tail*) of a file. These commands are the `head` and `tail` commands. They can be used as filters,or they can take a filename as an argument. By default they display the first (or last) 10 lines of the file or stream. Listing 9 uses the `dmesg` command to display bootup messages, in conjunction with `wc`, `tail`, and `head` to discover that there are 791 messages, then to display the last 10 of these, and finally to display the six messages starting 15 from the end. Some lines have been truncated in this output (indicated by ...).

Listing 9. Using wc, head, and tail to display boot messages

```
[ian@echidna lpi103-2]$ dmesg|wc
    791    5554   40186
[ian@echidna lpi103-2]$ dmesg | tail
input: HID 04b3:310b as /devices/pci0000:00/0000:00:1a.0/usb3/3-2/3-2.4/3-2.4:1.0/input/i
nput12
generic-usb 0003:04B3:310B.0009: input,hidraw1: USB HID v1.00 Mouse [HID 04b3:310b] on us
b-0000:00:1a.0-2.4/input0
usb 3-2.4: USB disconnect, address 11
usb 3-2.4: new low speed USB device using uhci_hcd and address 12
usb 3-2.4: New USB device found, idVendor=04b3, idProduct=310b
usb 3-2.4: New USB device strings: Mfr=0, Product=0, SerialNumber=0
usb 3-2.4: configuration #1 chosen from 1 choice
input: HID 04b3:310b as /devices/pci0000:00/0000:00:1a.0/usb3/3-2/3-2.4/3-2.4:1.0/input/i
nput13
generic-usb 0003:04B3:310B.000A: input,hidraw1: USB HID v1.00 Mouse [HID 04b3:310b] on us
b-0000:00:1a.0-2.4/input0
usb 3-2.4: USB disconnect, address 12
[ian@echidna lpi103-2]$ dmesg | tail -n15 | head -n 6
usb 3-2.4: USB disconnect, address 10
usb 3-2.4: new low speed USB device using uhci_hcd and address 11
usb 3-2.4: New USB device found, idVendor=04b3, idProduct=310b
usb 3-2.4: New USB device strings: Mfr=0, Product=0, SerialNumber=0
usb 3-2.4: configuration #1 chosen from 1 choice
input: HID 04b3:310b as /devices/pci0000:00/0000:00:1a.0/usb3/3-2/3-2.4/3-2.4:1.0/input/i
nput12
```

Another common use of `tail` is to *follow* a file using the `-f` option, usually with a line count of 1. You might use this when you have a background process that is generating output in a file and you want to check in and see how it is doing. In this mode, `tail` will run until you cancel it (using **Ctrl-c**), displaying lines as they are written to the file.

Expand, unexpand, and tr

When we created our text1 and text2 files, we created text2 with tab characters. Sometimes you may want to swap tabs for spaces or vice versa. The `expand` and `unexpand` commands do this. The `-t` option for both commands allows you to set the tab stops. A single value sets repeated tabs at that interval. Listing 10 shows how to expand the tabs in text2 to single spaces and another whimsical sequence of `expand` and `unexpand` that unaligns the text in text2.

Listing 10. Using expand and unexpand

```
[ian@echidna lpi103-2]$ expand -t 1 text2
9 plum
3 banana
10 apple
[ian@echidna lpi103-2]$ expand -t8 text2|unexpand -a -t2|expand -t3
9           plum
3           banana
10       apple
```

Unfortunately, you cannot use `unexpand` to replace the spaces in text1 with tabs, as `unexpand` requires at least two spaces to convert to tabs. However, you can use the `tr` command, which translates characters in one set (*set1*) to corresponding characters in another set (*set2*). Listing 11 shows how to use `tr` to translate spaces to tabs. Since `tr` is purely a filter, you generate input for it using the `cat` command. This example also illustrates the use of - to signify standard input to `cat`, so we can concatenate the output of `tr` and the text2 file.

Listing 11. Using tr

```
[ian@echidna lpi103-2]$ cat text1 |tr ' ' '\t'|cat - text2
1       apple
2       pear
3       banana
9       plum
3       banana
10      apple
```

If you are not sure what is happening in the last two examples, try using `od` to terminate each stage of the pipeline in turn; for example,
`cat text1 |tr ' ' '\t' | od -tc`

Pr, nl, and fmt

The `pr` command is used to format files for printing. The default header includes the filename and file creation date and time, along with a page number and two lines of blank footer. When output is created from multiple files or the standard input stream, the current date and time are used instead of the filename and creation date. You can print files side-by-side in columns and control many aspects of formatting through options. As usual, refer to the man page for details.

The `nl` command numbers lines, which can be convenient when printing files. You can also number lines with the `-n` option of the `cat` command. Listing 12 shows how to print our text file, and then how to number text2 and print it side-by-side with text1.

Listing 12. Numbering and formatting for print

```
[ian@echidna lpi103-2]$ pr text1 | head


2009-08-11 14:02                     text1                      Page 1


1 apple
2 pear
3 banana


[ian@echidna lpi103-2]$ nl text2 | pr -m - text1 | head


2009-08-11 15:36                                                Page 1


     1  9       plum                        1 apple
     2  3       banana                      2 pear
     3  10      apple                       3 banana
```

Another useful command for formatting text is the `fmt` command, which formats text so it fits within margins. You can join several short lines as well as split long ones. In Listing 13, we create text3 with a single long line of text using variants of the `!#:*` history feature to save typing our sentence four times. We also create text4 with one word per line. Then we use `cat` to display them unformatted including a displayed '$' character to show line endings. Finally, we use `fmt` to format them to a maximum width of 60 characters. Again, consult the man page for details on additional options.

Listing 13. Formatting to a maximum line length

```
[ian@echidna lpi103-2]$ echo "This is a sentence. " !#:* !#:1->text3
echo "This is a sentence. " "This is a sentence. " "This is a sentence. ">text3
[ian@echidna lpi103-2]$ echo -e "This\nis\nanother\nsentence.">text4
[ian@echidna lpi103-2]$ cat -et text3 text4
This is a sentence.  This is a sentence.  This is a sentence. $
This$
is$
another$
sentence.$
[ian@echidna lpi103-2]$ fmt -w 60 text3 text4
This is a sentence.  This is a sentence.  This is a
sentence.
This is another sentence.
```

Sort and uniq

The `sort` command sorts the input using the collating sequence for the locale (LC_COLLATE) of the system. The `sort` command can also merge already sorted files and check whether a file is sorted or not.

Listing 14 illustrates using the `sort`command to sort our two text files after translating blanks to tabs in text1. Since the sort order is by character, you might be surprised at the results. Fortunately, the `sort` command can sort by numeric values or by character values. You can specify this choice for the whole record or for each *field*. Unless you specify a different field separator,

fields are delimited by blanks or tabs. The second example in Listing 14 shows sorting the first field numerically and the second by collating sequence (alphabetically). It also illustrates the use of the `-u` option to eliminate any duplicate lines and keep only lines that are unique.

Listing 14. Character and numeric sorting

```
[ian@echidna lpi103-2]$ cat text1 | tr ' ' '\t' | sort - text2
10      apple
1       apple
2       pear
3       banana
3       banana
9       plum
[ian@echidna lpi103-2]$ cat text1|tr ' ' '\t'|sort -u -k1n -k2 - text2
1       apple
2       pear
3       banana
9       plum
10      apple
```

Notice that we still have two lines containing the fruit "apple", because the uniqueness test is performed on all of the two sort keys, k1n and k2 in our case. Think about how to modify or add steps to the above pipeline to eliminate the second occurrence of 'apple'.

Another command called `uniq` gives us another way of controlling the elimination of duplicate lines. The `uniq` command normally operates on sorted files, and removes **consecutive** identical lines from any file, whether sorted or not. The `uniq` command can also ignore some fields. Listing 15 sorts our two text files using the second field (fruit name) and then eliminates lines that are identical, starting at the second field (that is, we skip the first field when testing with `uniq`).

Listing 15. Using uniq

```
[ian@echidna lpi103-2]$ cat text1|tr ' ' '\t'|sort -k2 - text2|uniq -f1
10      apple
3       banana
2       pear
9       plum
```

Our sort was by collating sequence, so `uniq` gives us the "10 apple" line instead of the "1 apple". Try adding a numeric sort on key field 1 to see how to change this.

Cut, paste, and join

Now let's look at three more commands that deal with fields in textual data. These commands are particularly useful for dealing with tabular data. The first is the `cut` command, which extracts fields from text files. The default field delimiter is the tab character. Listing 16 uses `cut` to separate the two columns of text2 and then uses a space as an output delimiter, which is an exotic way of converting the tab in each line to a space.

Listing 16. Using cut

```
[ian@echidna lpi103-2]$ cut -f1-2 --output-delimiter=' ' text2
9 plum
3 banana
10 apple
```

The `paste` command pastes lines from two or more files side-by-side, similar to the way that the `pr` command merges files using its `-m` option. Listing 17 shows the result of pasting our two text files.

Listing 17. Pasting files

```
[ian@echidna lpi103-2]$ paste text1 text2
1 apple 9       plum
2 pear  3       banana
3 banana        10      apple
```

These examples show simple pasting, but `paste` can paste data from one or more files in several other ways. Consult the man page for details.

Our final field-manipulating command is `join`, which joins files based on a matching field. The files should be sorted on the join field. Since text2 is not sorted in numeric order, we could sort it and then join would join the two lines that have a matching join field (the field with value 3 in this case).

Listing 18. Joining files with join fields

```
[ian@echidna lpi103-2]$ sort -n text2|join -j 1 text1 -
3 banana banana
join: file 2 is not in sorted order
```

So what went wrong? Remember what you learned about character and numeric sorting back in the Sort and uniq section. The join is performed on matching characters according to the locale's collating sequence. It does not work on numeric fields unless the fields are all the same length.

We used the `-j 1` option to join on field 1 from each file. The field to use for the join may be specified separately for each file. You could, for example, join based on field 3 from one file and field 10 from another.

Let's also create a new file, text5, by sorting text 1 on the second field (the fruit name) and then replacing spaces with tabs. If we then sort text2 on its second field and join that with text 5 using the second field of each file as the join field, we should have two matches (apple and banana). Listing 19 illustrates this join.

Listing 19. Joining files with join fields

```
[ian@echidna lpi103-2]$ sort -k2 text1|tr ' ' '\t'>text5
[ian@echidna lpi103-2]$ sort -k2 text2 | join -1 2 -2 2 text5 -
apple 1 10
banana 3 3
```

Sed

Sed is the *s*tream *ed*itor. Several developerWorks articles, as well as many books and book chapters, are available on sed (see Resources). Sed is extremely powerful, and the tasks it can accomplish are limited only by your imagination. This small introduction should whet your appetite for sed, but is not intended to be complete or extensive.

As with many of the text commands we have looked at so far, sed can work as a filter or take its input from a file. Output is to the standard output stream. Sed loads lines from the input into the

pattern space, applies sed editing commands to the contents of the pattern space, and then writes the pattern space to standard output. Sed may combine several lines in the pattern space, and it may write to a file, write only selected output, or not write at all.

Sed uses regular expression syntax to search for and replace text selectively in the pattern space as well as to control which lines of text should be operated on by sets of editing commands. Regular expressions are covered more fully in the article on *searching text files using regular expressions* (which you can find in the series roadmap). A *hold buffer* provides temporary storage for text. The hold buffer may replace the pattern space, be added to the pattern space, or be exchanged with the pattern space. Sed has a limited set of commands, but these combined with regular expression syntax and the hold buffer make for some amazing capabilities. A set of sed commands is usually called a *sed script*.

Listing 20 shows three simple sed scripts. In the first one, we use the `s` (substitute) command to substitute an upper case for a lower case 'a' on each line. This example replaces only the first 'a', so in the second example, we add the 'g' (for global) flag to cause sed to change all occurrences. In the third script, we introduce the `d` (delete) command to delete a line. In our example, we use an address of 2 to indicate that only line 2 should be deleted. We separate commands using a semi-colon (;) and use the same global substitution that we used in the second script to replace 'a' with 'A'.

Listing 20. Beginning sed scripts

```
[ian@echidna lpi103-2]$ sed 's/a/A/' text1
1 Apple
2 peAr
3 bAnana
[ian@echidna lpi103-2]$ sed 's/a/A/g' text1
1 Apple
2 peAr
3 bAnAnA
[ian@echidna lpi103-2]$ sed '2d;$s/a/A/g' text1
1 apple
3 bAnAnA
```

In addition to operating on individual lines, sed can operate on a range of lines. The beginning and end of the range is separated by a comma (,) and may be specified as a line number, a regular expression, or a dollar sign ($) for end of file. Given an address or a range of addresses, you can group several commands between curly braces, { and }, to have these commands operate only on lines selected by the range. Listing 21 illustrates two ways of having our global substitution applied to only the last two lines of our file. It also illustrates the use of the `-e` option to add multiple commands to the script.

Listing 21. Sed addresses

```
[ian@echidna lpi103-2]$ sed -e '2,${' -e 's/a/A/g' -e '}' text1
1 apple
2 peAr
3 bAnAnA
[ian@echidna lpi103-2]$ sed -e '/pear/,/bana/{' -e 's/a/A/g' -e '}' text1
1 apple
2 peAr
3 bAnAnA
```

Sed scripts may also be stored in files. In fact, you will probably want to do this for frequently used scripts. Remember earlier we used the `tr` command to change blanks in text1 to tabs. Let's now do that with a sed script stored in a file. We will use the `echo` command to create the file. The results are shown in Listing 22.

Listing 22. A sed one-liner

```
[ian@echidna lpi103-2]$ echo -e "s/ /\t/g">sedtab
[ian@echidna lpi103-2]$ cat sedtab
s/ /    /g
[ian@echidna lpi103-2]$ sed -f sedtab text1
1       apple
2       pear
3       banana
```

There are many handy sed one-liners such as Listing 22. See Resources for links to some.

Our final sed example uses the `=` command to print line numbers and then filter the resulting output through sed again to mimic the effect of the `nl` command to number lines. Listing 23 uses `=` to print line numbers, then uses the `N` command to read a second input line into the pattern space, and finally removes the newline character (\n) between the two lines in the pattern space.

Listing 23. Numbering lines with sed

```
[ian@echidna lpi103-2]$ sed '=' text2
1
9       plum
2
3       banana
3
10      apple
[ian@echidna lpi103-2]$ sed '=' text2|sed 'N;s/\n//'
19      plum
23      banana
310     apple
```

Not quite what we wanted! What we would really like is to have our numbers aligned in a column with some space before the lines from the file. In Listing 24, we enter several lines of commands (note the > secondary prompt). Study the example and refer to the explanation below.

Listing 24. Numbering lines with sed - round two

```
[ian@echidna lpi103-2]$ cat text1 text2 text1 text2>text6
[ian@echidna lpi103-2]$ ht=$(echo -en "\t")
[ian@echidna lpi103-2]$ sed '=' text6|sed "N
> s/^/      /
> s/^.*\(......\)\n/\1$ht/"
     1  1 apple
     2  2 pear
     3  3 banana
     4  9       plum
     5  3       banana
     6  10      apple
     7  1 apple
     8  2 pear
     9  3 banana
    10  9       plum
    11  3       banana
    12  10      apple
```

Here are the steps we took:

1. We first used `cat` to create a twelve-line file from two copies each of our text1 and text2 files. There's no fun in formatting numbers in columns if we don't have differing numbers of digits.
2. The bash shell uses the tab key for command completion, so it can be handy to have a captive tab character that you can use when you want a real tab. We use the `echo` command to accomplish this and save the character in the shell variable 'ht'.
3. We create a stream containing line numbers followed by data lines as we did before and filter it through a second copy of sed.
4. We read a second line into the pattern space
5. We prefix our line number at the start of the pattern space (denoted by ^) with six blanks.
6. We then substitute all of the line up to the newline with the last six characters before the newline plus a tab character. This will align our line numbers in the first six columns of the output line. Note that the left part of the 's' command uses '\(' and '\)' to mark the characters that we want to use in the right part. In the right part, we reference the first such marked set (and only such set in this example) as \1. Note that our command is contained between double quotes ('') so that substitution will occur for $ht.

Version 4 of sed contains documentation in `info` format and includes many excellent examples. These are not included in the older version 3.02. GNU sed will accept `sed --version` to display the version.

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.
- Part 1, Part 2, and Part 3 of the series *Sed by example* are a great way to develop your sed skills.
- In the developerWorks Linux zone, find more resources for Linux developers, and scan our most popular articles and tutorials.
- See all Linux tips and Linux tutorials on developerWorks.
- Stay current with developerWorks technical events and Webcasts.

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; with your personal profile and custom home page, you can tailor developerWorks to your interests and interact with other developerWorks users.

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.