Introduction to text manipulation on UNIX-based systems

Using standard utilities

Brad Yoes (wyoes@us.ibm.com)
Migration Engineer
IBM USA

14 March 2012

This introduction to text manipulation on UNIX platforms provides an overview of some common commands widely available and installed standard on most UNIX-based releases. Many times these standard utilities are ignored in favor of more modern text-processors such as Perl, Python, or Ruby, which are not always installed on a system. An introductory review of these tools helps practitioners who are learning UNIX or Linux or those who may be looking to renew forgotten knowledge.

Introduction

A basic tenets of UNIX philosophy is to create programs (or processes) that do one thing, and do that one thing well. It is a philosophy demanding careful thought about interfaces and ways of joining these smaller (hopefully more simple) processes together to create useful results. Normally textual data flows between these interfaces. Over time, more and more advanced text processing tools and languages have been developed. For languages, earlier on there was perl, later came python, and ruby. While these and other languages are very capable text processors, such tools are not always available, especially in a production environment. In this article, a number of basic UNIX text processing commands are demonstrated and may be used individually or in conjunction with each other to solve problems which may also be addressed with newer languages. For many people, an example provides more information than long winded explanations. Please note because of the variety of UNIX and UNIX-like systems available, command flags, program behavior, and output differs between implementations.

Use of cat

The `cat` command is one of the most basic commands. It is used to create, append, display, and concatenate files.

We can create a file with `cat` using the '>' to redirect standard input (`stdin`) to a file. Using the '>' operator truncates the contents of the output file specified. Text entered after that is redirected

to the file specified to the right of the '>' operator. The control-d signals an end-of-file, returning control to the shell.

Example of cat to create a file:

```
$ cat > grocery.list
apples
bananas
plums
<ctrl-d>
$
```

Use the '>>' operator to append standard input into an existing file.

Example of cat to append a file:

```
$ cat >> grocery.list
carrots
<ctrl-d>
```

Examine the contents of the grocery.list file, using `cat` without flags. Notice how the file contents include input from the redirection and append operator examples.

Example of cat without flags:

```
$ cat grocery.list
apples
bananas
plums
carrots
```

The `cat` command can be used to number the lines of a file.

Example of cat to count lines:

```
$ cat -n grocery.list
     1  apples
     2  bananas
     3  plums
     4  carrots
```

Use of nl

The `nl` filter reads lines from `stdin` or from specified files. Output is written to `stdout`, and may be redirected to a file or piped to another process. Behavior of `nl` is controlled through various command-line options.

By default, `nl` counts lines similar to `cat -n`.

Example default usage of nl:

```
$nl grocery.list
     1  apples
     2  bananas
     3  plums
     4  carrots
```

Use the -b flag to specify lines to be numbered. This flag takes as its argument a “type”. The type tells `nl` which lines need to be numbered – use ‘a’ to number all lines, ‘t’ tells `nl` to not number empty lines or lines that are only whitespace, ‘n’ specifies no lines be numbered. In the example a type of ‘p’ for pattern is shown. `nl` numbers the lines specified by a regular expression pattern, in this case, lines starting with the letters ‘a’ or ‘b’.

Example of nl to number lines conforming to a regex:

```
$ nl -b p^[ba] grocery.list
     1  apples
     2  bananas
       plums
       carrots
```

By default, `nl` separates the line number from the text using a tab. Use -s to specify a different delimiter, such as the ‘=’ sign.

Example of nl to specify a delimiter:

```
$nl –s= grocery.list
     1=apples
     2=bananas
     3=plums
     4=carrots
```

Use of wc

The `wc` (wordcount) command counts the number of lines, words (separated by whitespace), and characters in specified files, or from `stdin`.

Examples of wc usage:

```
$wc grocery.list
       4       4      29 grocery.list
$wc -l grocery.list
       4 grocery.list
$wc -w grocery.list
       4 grocery.list
$wc -c grocery.list
      29 grocery.list
```

Using grep

The `grep` command searches specified files or `stdin` for patterns matching a given expression(s). Output from `grep` is controlled by various option flags.

For demonstration, a new file was created to use with the grocery.list.

```
$cat grocery.list2
Apple Sauce
wild rice
black beans
kidney beans
dry apples
```

Example of basic grep usage:

```
$ grep apple grocery.list grocery.list2
grocery.list:apples
grocery.list2:dry apples
```

`grep` has a sizable number of option flags. Following are some examples demonstrating usage of a few options.

To display the filename (if multiple files) with number of lines on which the pattern was found – in this case, count the number of lines the word ‘apple’ occurs in each file.

Example of grep - counting number of matches in files:

```
$ grep -c apple grocery.list grocery.list2
grocery.list:1
grocery.list2:1
```

When searching multiple files, using the -h option suppresses printing the filename as part of the output.

Example of grep - suppress filename in output:

```
$ grep -h apple grocery.list grocery.list2
apples
dry apples
```

In many situations, a case-insensitive search is desired. The `grep` command has the -i option to ignore case-sensitivity when doing searches.

Example of grep – case insensitive:

```
$ grep -i apple grocery.list grocery.list2

grocery.list:apples
grocery.list2:Apple Sauce
grocery.list2:dry apples
```

Sometimes, only the filename needs to be printed, not the pattern-matching line. `grep` provides the -l option to only print filenames containing lines with a matching pattern.

Example of grep – filenames only:

```
$ grep -l carrot grocery.list grocery.list2
grocery.list
```

Line numbers can be provided as part of the output. Use the -n option to include line numbers.

Example of grep – include line numbers:

```
$ grep -n carrot grocery.list grocery.list2
grocery.list:4:carrots
```

There are times when lines not matching the pattern are the desired output. In such situations use the -v option.

Example of grep – inverted matching:

```
$ grep -v beans grocery.list2
Apple Sauce
wild rice
dry apples
```

Sometimes the pattern desired forms a “word” surrounded by whitespace or other characters like a dash or parentheses. Most versions of `grep` provide a -w option to ease writing searches for these patterns.

Example of grep – word matching:

```
$ grep -w apples grocery.list grocery.list2
grocery.list:apples
grocery.list2:dry apples
```

Streams, pipes, redirects, tee, here docs

In UNIX a terminal by default contains three streams, one for input, and two output-based streams. The input stream is referred to as `stdin`, and is generally mapped to the keyboard (other input devices may be used, or could be piped from another process). The standard output stream is referred to as `stdout`, and generally prints to the terminal, or output may be consumed by another process (as `stdin`). The other output stream `stderr` primarily used for status reporting usually prints to the terminal like `stdout`. As each of these streams has its own file descriptor, each may be piped or redirected separately from the other, even if they are all connected to the terminal. File descriptors for each of these streams are:

- stdin = 0
- stdout = 1
- stderr = 2

These streams can be piped and redirected to files or other processes. This construct is commonly referred to as building a pipeline. For example, a programmer may want to merge the `stdout` and `stderr` streams, and then display them on the terminal but also save the results to a file to examine build issues. Using `2>&1` the `stderr` stream with file descriptor `2` is redirected to `&1`, a ‘pointer’ the `stdout` stream. This effectively merges `stderr` into `stdout`. Using the ‘|’ symbol indicates a pipe. A pipe links `stdout` from the left-hand process (`make`) to `stdin` of the right-hand process (`tee`). The `tee` command duplicates the (merged) `stdout` stream sending the data to the terminal and to a file, in this example, called build.log.

Example of merging and splitting standard streams:

```
$ make –f build_example.mk 2>&1 | tee build.log
```

Another example of redirection, a copy of a text file is made using the cat command and some stream redirection.

Example of redirection to make a backup file:

```
$ cat < grocery.list > grocery.list.bak
```

Earlier the `nl` command was used to add line numbers to a file displayed on `stdout`. The pipe can be used to send the `stdout` stream (from cat grocery.list) to another process, in this case, the `nl` command.

Example simple pipe to nl:

```
$ cat grocery.list | nl
     1  apples
     2  bananas
     3  plums
     4  carrots
```

Another example shown earlier was to do a case-insensitive search of a file for a pattern. This can be done using redirection - in this case from `stdin`, or using a pipe, similar to the simple pipe example above.

Example with grep - stdin redirection and pipe:

```
$ grep -i apple < grocery.list2
Apple Sauce
dry apples
$cat grocery.list2 | grep -i apple
Apple Sauce
dry apples
```

In some situations a block of text will be redirected into a command or file as part of a script. A mechanism to accomplish this is to use a ‘here document’ or ‘here-doc’. To embed a here-doc into a script, the ‘`<<`’ operator is used to redirect the following text, until a end-of-file delimiter is reached. The delimiter is specified after the `<<` operator.

Example basic here-doc on the command line:

```
$ cat << EOF
> oranges
> mangos
> pinapples
> EOF
oranges
mangos
pinapples
```

This output can be redirected to a file, in this example the delimiter changed from ‘EOF’ to ‘!’. Then `tr` command (explained later) is used to upper-case the letters with a here-doc.

Example basic here-doc redirected to a file:

```
cat << ! > grocery.list3
oranges
mangos
pinapples
!
$ cat grocery.list3
oranges
mangos
pinapples
$tr [:lower:] [:upper:] << !
> onions
> !
ONIONS
```

Using head and tail

The `head` and `tail` commands are used to examine the top (head) or bottom (tail) parts of files. To display the top two lines and bottom two lines of a file use the -n option flag with these commands respectively. Similarly, the -c option displays the first or last characters in the file.

Example basic use of head and tail commands:

```
$ head -n2 grocery.list
apples
bananas
$ tail -n2 grocery.list
plums
carrots
$ head -c12 grocery.list
apples
banan
$ tail -c12 grocery.list
ums
carrots
```

A common use for the `tail` command is to watch log files or the output of running processes to see if there are issues, or to note when a process finishes. The -f (`tail –f`) option causes tail to continue to watch the stream, even after the end-of-file marker is reached, and continue displaying output when the stream contains more data.

Using tr

The `tr` command is used to translate characters from `stdin`, displaying them on `stdout`. In its general form, `tr` takes two sets of characters, and replaces characters from the first set with characters from the second set. A number of pre-defined character classes (sets) are available to be used by `tr`, and some other commands.

These classes are:

- alnum - alphanumeric characters
- alpha - alphabetic characters
- blank - whitespace characters
- cntrl - control characters
- digit - numeric characters
- graph - graphic characters
- lower - lower-case alphabetic characters
- print - printable characters
- punct - punctuation characters
- space - space characters
- upper - upper-case characters
- xdigit - hexadecimal characters

The `tr` command can translate lowercase characters in a string to upper case.

Example tr - upper-case a string:

```
$ echo "Who is the standard text editor?" |tr [:lower:] [:upper:]
WHO IS THE STANDARD TEXT EDITOR?
```

`tr` can be used to delete named characters from a string.

Example tr - delete characters from a string:

```
$ echo 'ed, of course!' |tr -d aeiou
d, f crs!
```

Use `tr` to translate any named characters in a string to a space. When multiple named characters are encountered in sequence, they are translated into a single space.

Behavior of the -s option flag differs between systems.

Example tr - translate characters to a space:

```
$ echo 'The ed utility is the standard text editor.' |tr -s astu ' '
The ed ili y i he nd rd ex edi or.
```

The -s option flag can be used to suppresses extra white space in a sting.

```
$ echo 'extra     spaces – 5’ | tr -s [:blank:]
extra spaces - 5
$ echo ‘extra  tabs – 2’ | tr -s [:blank:]
extra   tabs – 2
```

A common problem when transferring files between UNIX and Windows based systems is line delimiters. On UNIX systems the delimiter is a new line, while Windows systems use two characters, a carriage return followed by a newline. Using `tr` with some redirection is one way to fix this formatting problem.

Example tr - remove carriage returns:

```
$ tr -d '\r' < dosfile.txt > unixfile.txt
```

Use of colrm

Using `colrm` columns of text can be cut from a stream. In the first example, `colrm` is used to cut from column 4 to the end of line for each line off the pipe. Next, the same file is sent to `colrm` to remove columns 4 through 5.

Example colrm to remove columns:

```
$ cat grocery.list |colrm 4
app
ban
plu
car
$ cat grocery.list |colrm 4 5
apps
banas
plu
carts
```

Use of expand and unexpand

The `expand` command changes tabs to spaces, while `unexpand` changes spaces to tabs. These commands take input off of `stdin` or from files named on the command line. Using the -t option, one or more tab stop can be set.

Example of expand and unexpand:

```
$ cat grocery.list|head -2|nl|nl
     1       1  apples
     2       2  bananas
$ cat grocery.list|head -2|nl|nl|expand -t 5
     1         1    apples
     2         2    bananas
$ cat grocery.list|head -2|nl|nl|expand -t 5,20
     1                   1 apples
     2                   2 bananas
$ cat grocery.list|head -2|nl|nl|expand -t 5,20|unexpand -t 1,5
                1                   1 apples
                2                   2 bananas
```

Use of comm, cmp, and diff

To demonstrate these commands, two new files are created.

Create files for demonstration:

```
cat << EOF > dummy_file1.dat
011 IBM 174.99
012 INTC 22.69
013 SAP 59.37
014 VMW 102.92
EOF
cat << EOF > dummy_file2.dat
011  IBM 174.99
012 INTC 22.78
013 SAP 59.37
014 vmw 102.92
EOF
```

The `diff` command compares two files, reporting on differences between them. `diff` takes a number of option flags. In the following example, a default `diff` is shown first, followed by a `diff` using the -w option which ignores whitespace, then finish with an example of the -i option flag which ignores differences between upper and lower case when doing comparisons.

Examples of diff command:

```
$ diff dummy_file1.dat dummy_file2.dat
1,2c1,2
< 011 IBM 174.99
< 012 INTC 22.69
---
> 011  IBM 174.99
> 012 INTC 22.78
4c4
< 014 VMW 102.92
---
> 014 vmw 102.92

$ diff -w dummy_file1.dat dummy_file2.dat
2c2
< 012 INTC 22.69
```

```
---
> 012 INTC 22.78
4c4
< 014 VMW 102.92
---
> 014 vmw 102.92

$ diff -i dummy_file1.dat dummy_file2.dat
1,2c1,2
< 011 IBM 174.99
< 012 INTC 22.69
---
> 011  IBM 174.99
> 012 INTC 22.78
```

The `comm` command compares two files, but behaves much differently than diff. `comm` produces three columns of output - lines only in file1 (column 1), lines only in file2 (column 2), and lines common to both files (column 3). Option flags can be used to suppress columns of output. This command is probably most useful to suppress column 1 and column 2, showing only lines common to both files, as shown below.

Example of comm command:

```
$ comm dummy_file1.dat dummy_file2.dat
        011  IBM 174.99
011 IBM 174.99
012 INTC 22.69
        012 INTC 22.78
                013 SAP 59.37
014 VMW 102.92
        014 vmw 102.92

$ comm -12 dummy_file1.dat dummy_file2.dat
013 SAP 59.37
```

The `cmp` command also compares two files. However, different than `comm` or `diff`, the `cmp` command (by default) reports the byte and line number where the two files first are different.

Example of cmp command:

```
$ cmp dummy_file1.dat dummy_file2.dat
dummy_file1.dat dummy_file2.dat differ: char 5, line 1
```

Using fold

Using the `fold` command, lines are broken at a specified width. Originally this command was used to help format text for fixed-width output devices incapable of wrapping text. The -w option flag allows specification of a line width to use instead of the default 80 columns.

Example using fold:

```
$ fold -w8 dummy_file1.dat
011 IBM
174.99
012 INTC
 22.69
013 SAP
59.37
014 VMW
102.92
```

Using paste

The `paste` command is used to align files side-by-side, merging records from each file respectively. Using redirection, new files can be created by joining each record of one file, with records in another file.

Create files for demonstration:

```
cat << EOF > dummy1.txt
IBM
INTC
SAP
VMW
EOF
cat << EOF > dummy2.txt
174.99
22.69
59.37
102.92
EOF
```

Example 1 of paste - lines from multiple files:

```
$ paste dummy1.txt dummy2.txt grocery.list
IBM     174.99  apples
INTC    22.69   bananas
SAP     59.37   plums
VMW     102.92  carrots
```

There is a -s option flag used to process the files one at a time (serially) instead of in parallel. Notice below, the columns align with the rows in the example above.

Example 2 of paste - lines from multiple files:

```
$ paste -s dummy1.txt dummy2.txt grocery.list
IBM     INTC    SAP     VMW
174.99  22.69   59.37   102.92
apples  bananas plums   carrots
```

If only one file is specified, or `paste` is processing `stdin`, by default the input is listed in one column. Using the -s option flag, the output is listed on one line. Since output is condensed to a single line, use a delimiter to separate the returned fields (tab is the default delimiter). In this example, the `find` command is used to locate directories where 64-bit libraries would likely be found, and builds a path suitable for appending to a $LD_LIBRARY_PATH variable.

Examples of paste - with delimiter:

```
$ find /usr -name lib64 -type d|paste -s -d:
/usr/lib/qt3/lib64:/usr/lib/debug/usr/lib64:/usr/X11R6/lib/X11/locale/lib64:/usr/X11R6/
lib64:/usr/lib64:/usr/local/ibm/gsk7_64/lib64:/usr/local/lib64

$ paste -d, dummy1.txt dummy2.txt
IBM,174.99
INTC,22.69
SAP,59.37
VMW,102.92
```

Using bc

For an easy way to do math in the shell, consider `bc`, the “basic calculator” or “bench calculator”. Some shells offer ways to do math natively, others rely on `expr` to evaluate expressions. Using `bc`, calculations can be portable across shells and UNIX systems, just be careful around vendor extensions.

Example of bc – simple calculations:

```
$ echo 2+3|bc
5

$ echo 3*3+2|bc
11

$ VAR1=$(echo 2^8|bc)
$ echo $VAR1
256

$ echo "(1+1)^8"|bc
256
```

`bc` can perform more than these simple calculations. It is an interpreter defining its own internal and user defined functions, syntax, and flow control, similar to a programming language. By default `bc` does not include any digits to the right of the decimal point. Increase precision in the output by using the special `scale` variable. As the example shows, `bc` scales to large numbers, and carries out to lengthy precisions. Use `obase` or `ibase` to control conversion base for input and output numbers. In the example below:

- `obase` changes the default output base (ten), converting the result to hexadecimal
- for this square root of 2 approximation, `scale` specifies number of digits to the right of the decimal
- support for large numbers is illustrated by calculating 2 to the 128th
- the `sqrt()` internal function is called to calculate square root of 2
- From `ksh`, compute and print a percentage

Example of bc – more calculations:

```
$ echo "obase=16; 2^8-1"|bc
FF

$ echo "99/70"|bc
1

$ echo "scale=20; 99/70"|bc
1.41428571428571428571

$ echo "scale=20;sqrt(2)"|bc
1.41421356237309504880

$ echo 2^128|bc
340282366920938463463374607431768211456

$ printf "Percentage: %2.2f%%\n" $(echo .9963*100|bc)
Percentage: 99.63%
```

The man page for `bc` is detailed and includes examples.

Using split

A useful task for the `split` command is to break large datafiles into smaller files for processing. In this example, BigFile.dat is shown to have 165782 lines using the `wc` command. The -l option flag tells `split` the maximum number of lines for each output file. `split` allows for a prefix to be specified for output filenames, below BigFile_ is the specified prefix. Other options allow for suffix control, and on BSD, a -p option flag allows for splits to occur at a regular expression like the `csplit` (context split) command. Take a look at the man page for more information.

Example of split:

```
$ wc BigFile.dat
165782  973580 42557440 BigFile.dat

$ split -l 15000 BigFile.dat BigFile_

$ wc BigFile*
  165782  973580 42557440 BigFile.dat
   15000   87835 3816746 BigFile_aa
   15000   88483 3837494 BigFile_ab
   15000   89071 3871589 BigFile_ac
   15000   88563 3877480 BigFile_ad
   15000   88229 3855486 BigFile_ae
    7514   43817 1908914 BigFile_af
  248296 1459578 63725149 total
```

Using cut

The `cut` command is used to ‘trim out’ column-based sections of a file or data piped to it from `stdin`. It cuts data by bytes (-b), characters (-c), or fields (-f) as specified by a list. The lists of fields or byte/character positions are specified using comma separated lists and hyphens. Just specify the position or field desired if only one is needed for output. A range of fields can be specified using a hyphen such that 1-3 prints fields (or positions) 1 through 3, -2 prints from the beginning of the line up to field 2 (or byte/character 2), and 3- specifies `cut` to print field (or position 3) through the end of line. Multiple fields can be specified using a comma. Some other useful flags are -d to specify a field delimiter, and -s, to suppress lines without delimiters.

Examples of cut:

```
$ cat << EOF > dummy_cut.dat
# this is a data file
ID,Name,Score
13BA,John Smith,100
24BC,Mary Jones,95
34BR,Larry Jones,94
36FT,Joe Ruiz,93
40RM,Kay Smith,91
EOF

$ cat dummy_cut.dat |cut -d, -f1,3
# this is a data file
ID,Score
13BA,100
24BC,95
```

```
34BR,94
36FT,93
40RM,91

$ cat dummy_cut.dat |cut -b6-
s is a data file
me,Score
John Smith,100
Mary Jones,95
Larry Jones,94
Joe Ruiz,93
Kay Smith,91

$ cat dummy_cut.dat |cut -f1- -d, -s
ID,Name,Score
13BA,John Smith,100
24BC,Mary Jones,95
34BR,Larry Jones,94
36FT,Joe Ruiz,93
40RM,Kay Smith,91
```

Using uniq

The `uniq` command is typically used to uniquely list lines from an input source (usually a file or `stdin`). To operate properly, duplicate lines must be contiguously positioned in the input. Normally input to the `uniq` command is sorted, so duplicate lines are aligned. Two common flags used with the `uniq` command are -c to print the count of the number of times each line appeared, and -d can be utilized to display one instance of duplicate lines.

Examples of uniq:

```
$ cat << EOF > dummy_uniq.dat

13BAR   Smith   John    100
13BAR   Smith   John    100
24BC    Jone    Mary    95
34BRR   Jones   Larry   94
36FT    Ruiz    Joe     93
40REM   Smith   Kay     91
13BAR   Smith   John    100
99BAR   Smith   John    100
13XIV   Smith   Cindy   91


EOF

$ cat dummy_uniq.dat | uniq

13BAR   Smith   John    100
24BC    Jone    Mary    95
34BRR   Jones   Larry   94
36FT    Ruiz    Joe     93
40REM   Smith   Kay     91
13BAR   Smith   John    100
99BAR   Smith   John    100
13XIV   Smith   Cindy   91

$ cat dummy_uniq.dat | sort |uniq

13BAR   Smith   John    100
13XIV   Smith   Cindy   91
24BC    Jone    Mary    95
34BRR   Jones   Larry   94
```

```
36FT    Ruiz    Joe     93
40REM   Smith   Kay     91
99BAR   Smith   John    100

$ cat dummy_uniq.dat | sort |uniq -d

13BAR   Smith   John    100

$ cat dummy_uniq.dat | sort |uniq -c
   3
   3 13BAR   Smith   John    100
   1 13XIV   Smith   Cindy   91
   1 24BC    Jone    Mary    95
   1 34BRR   Jones   Larry   94
   1 36FT    Ruiz    Joe     93
   1 40REM   Smith   Kay     91
   1 99BAR   Smith   John    100
```

Using sort

To arrange rows in `stdin` or a file in a particular order, such as alphabetic or numeric, the `sort` command may be used. By default output from `sort` is written on `stdout`. Environment variables such as LC_ALL, LC_COLLATE, or LANG can affect the output of `sort` and other commands. Notice how the example file shows 2 separate duplicate records – one dupe for IBM, the other dupe is a blank line.

Example of sort – default behavior:

```
$ cat << EOF > dummy_sort1.dat

014 VMW, 102.92
013 INTC, 22.69
012 sap,  59.37
011 IBM, 174.99
011 IBM, 174.99

EOF

$ sort dummy_sort1.dat


011 IBM, 174.99
011 IBM, 174.99
012 sap,  59.37
013 INTC, 22.69
014 VMW, 102.92
```

`sort` has a great flag that can substitute in place of the `uniq` command in many circumstances. The -u option flag sorts the file, removing duplicate rows so a listing of unique rows of output is produced.

Example of sort – sort unique:

```
$ sort -u dummy_sort1.dat

011 IBM, 174.99
012 sap,  59.37
013 INTC, 22.69
014 VMW, 102.92
```

Sometimes, the reverse ordering of input is desired. By default `sort` arranges the order from smallest to largest (for numeric) and in alphabetical order for character data. Use the -r option flag to reverse the default sorting order.

Example of sort – sort reverse order:

```
$ sort -ru dummy_sort1.dat
014 VMW, 102.92
013 INTC, 22.69
012 sap,  59.37
011 IBM, 174.99
```

Different situations call for a file to be sorted on certain fields or “keys”. Fortunately `sort` has the -k option flag allowing a sort key to be specified by position. Fields are separated by whitespace by default.

Example of sort – sort with a key:

```
$ sort -k2 -u dummy_sort1.dat

011 IBM, 174.99
013 INTC, 22.69
014 VMW, 102.92
012 sap,  59.37
```

When case sensitivity is an issue, `sort` provides the -f option flag, which ignores case when doing comparisons. When combining multiple flags as shown below, some versions of UNIX need these flags specified in a different order.

Example of sort – sort ignoring case:

```
$ sort -k2 -f -u dummy_sort1.dat

011 IBM, 174.99
013 INTC, 22.69
012 sap,  59.37
014 VMW, 102.92
```

So far, all the sorts have been the alphabetic variety. When data needs to be sorted in numeric order, use the -n option flag.

Example of sort – numeric sort:

```
$ sort -n -k3 -u dummy_sort1.dat

013 INTC, 22.69
012 sap,  59.37
014 VMW, 102.92
011 IBM, 174.99
```

Some inputs may use characters other than white space to separate fields in a row. Use the -t option flag to specify a non-default delimiter such as a comma.

Example of sort – sort on field using non-default delimiter:

```
$ sort -k2 -t"," -un dummy_sort1.dat

013 INTC, 22.69
012 sap,  59.37
014 VMW, 102.92
011 IBM, 174.99
```

Using join

Anyone familiar with writing database queries recognizes the utility of the `join` command. Like most UNIX commands, output is displayed on `stdout`. To “join” files together, specified fields from two file are compared on a line by line basis. If no fields are specified, `join` matches on fields from the beginning of each line. The default field separator is whitespace (some systems it’s simply a space or adjacent spaces). When a field match occurs, one line of output is written for each pair of lines with matching fields. For legitimate results, both files should be sorted on the fields to be matched. Not all systems implement `join` the same way.

This example uses -t to specify a field separator, and demonstrates joining two files on the first field (default) delimited with commas. Database operators would recognize it as an inner join, displaying only matching rows.

Example of join – using non-default field delimiter:

```
cat << EOF > dummy_join1.dat
011,IBM,Palmisano
012,INTC,Otellini
013,SAP,Snabe
014,VMW,Maritz
015,ORCL,Ellison
017,RHT,Whitehurst
EOF

cat << EOF > dummy_join2.dat
011,174.99,14.6
012,22.69,10.4
013,59.37,26.4
014,102.92,106.1
016,27.77,31.2
EOF

cat << EOF > dummy_join3.dat
IBM,Armonk
INTC,Santa Clara
SAP,Walldorf
VMW,Palo Alto
ORCL,Redwood City
EMC,Hopkinton
EOF

$ join -t, dummy_join1.dat dummy_join2.dat
011,IBM,Palmisano,174.99,14.6
012,INTC,Otellini,22.69,10.4
013,SAP,Snabe,59.37,26.4
014,VMW,Maritz,102.92,106.1
```

To specify fields on which to“join” in each file the -j[1,2] x option flag (or simply -1 x or -2 x) can be used. Option flag -j1 2 or -1 2 specifies the second field of file one, the first file listed on the

command. This example shows how to join the files based on field 1 in the first file and field 2 in second file, also an inner join only matching rows.

Example of join – specified fields:

```
$ join -t, -j1 1 -j2 2 dummy_join3.dat dummy_join1.dat
IBM,Armonk,011,Palmisano
INTC,Santa Clara,012,Otellini
SAP,Walldorf,013,Snabe
VMW,Palo Alto,014,Maritz
ORCL,Redwood City,015,Ellison
```

Keeping with spirit of database related examples, flags can be used to produce a left outer join. A left outer join includes all rows from the left first file or table and matching rows from the second file or table. Use -a to include all rows from the specified file.

Example of join – left outer join:

```
$ join -t, -a1 dummy_join1.dat dummy_join2.dat
011,IBM,Palmisano,174.99,14.6
012,INTC,Otellini,22.69,10.4
013,SAP,Snabe,59.37,26.4
014,VMW,Maritz,102.92,106.1
015,ORCL,Ellison
017,RHT,Whitehurst
```

Full outer joins include all rows from both files or tables, it does not matter if the fields match. A full outer join can be done by specifying both files with the -a option flag.

Example of join – full outer join:

```
$ join -t, -a1 -a2 -j1 2 -j2 1 dummy_join1.dat dummy_join3.dat
IBM,011,Palmisano,Armonk
INTC,012,Otellini,Santa Clara
SAP,013,Snabe,Walldorf
VMW,014,Maritz,Palo Alto
ORCL,015,Ellison,Redwood City
EMC,Hopkinton
017,RHT,Whitehurst
```

Using sed

The stream editor `sed` is a useful text parsing and manipulation utility handy for doing transformations on files or streams of data. It reads in text one line at a time, applying the specified commands on the line of text. By default output goes to `stdout`. Commands `sed` uses perform operations such as deleting text from a buffer, appending or inserting text into a buffer, writing to a file, transforming text based on regular expressions, and more.

A basic example of `sed` substitution shows the -e option flag being used to specify the expression or the edit script. Multiple expressions or edits can be specified for a single `sed` execution. Notice the components of the `sed` edit. The “s” at the front of the edit indicates this is a substitution command. Using a “/” as a delimiter, the pattern “IBM” to replace is indicated first. Next, the replacement pattern appears between two “/” delimiters. Last, the “g” indicates to make the change globally in the current text buffer. The third demonstration in this example illustrates a combination

of three edits: replace backslashes with slashes, spaces with underscores, and remove colon characters – notice how the backslash "\" characters are escaped.

Example of sed – basic substitution / multiple edits:

```
$ echo "IBM 174.99" |sed –e 's/IBM/International Business Machines/g'
International Business Machines 174.99

$ echo "Oracle DB"|sed -e 's/Oracle/IBM/g' -e 's/DB/DB2/g'
IBM DB2

$ echo "C:\Program Files\PuTTY\putty.exe"| sed -e 's/\\/\//g' -e 's/ /_/g' -e 's/://g'
C/Program_Files/PuTTY/putty.exe
```

In the following example a file is set up to demonstrate another feature of `sed`. Besides substitution, filtering is another frequent use of `sed`. The UNIX `grep` command is a commonly employed filter, it is not unusual to discover multiple ways to manipulate text on the command line. This example shows how to use the `sed` delete command to remove lines beginning with "#" or whitespace then "#". An example of `grep` utilizing the same pattern is shown for reference.

Example of sed - filtering:

```
cat << EOF > dummy_sed.txt
# top of file
  # the next line here
# Last Name, Phone
Smith, 555-1212
Jones, 555-5555 # last number
EOF

$ sed '/^[[:space:]]*#/d' dummy_sed.txt
Smith, 555-1212
Jones, 555-5555 # last number

$ grep -v  ^[[:space:]]*# dummy_sed.txt
Smith, 555-1212
Jones, 555-5555 # last number
```

To better understand `sed` behavior a few more patterns are demonstrated. A file is created so the patterns can act upon some text. The first `sed` pattern shows how to remove the last 4 characters from strings (filenames) listed in the file. Next, the pattern removes all characters to the right of the dot "." indicating a file extension. A pattern to remove blank lines is shown. A special character, the ampersand "&" allows the search pattern to be used as part of the output. In this example, IBM is part of the input pattern, and is specified as part of the output using the ampersand. The last pattern demonstrated in this series shows how `sed` can be used to remove carriage returns from a text file transferred from a Windows-based system. The "^M" is entered into the script or on the command line by pressing first the control-v, then pressing control-m. Please note terminal characteristics may affect entry of the control-v, control-m combination.

Example of sed – more patterns:

```
cat << EOF > filelist.txt
PuTTY.exe

sftp.exe
```

```
netstat.exe
servernames.list
EOF

$ sed 's/....$//' filelist.txt
PuTTY

sftp
netstat
servernames.

$ sed 's/\..*$//g' filelist.txt
PuTTY

sftp
netstat
servernames

$ sed '/^$/d' filelist.txt
PuTTY.exe
sftp.exe
netstat.exe
servernames.list

$ echo "IBM 174.99" |sed 's/IBM/&-International Business Machines/g'
IBM-International Business Machines 174.99

$ cat dosfile.txt | sed 's/^M//' > unixfile.txt
```

`sed` commands can operate on specified address ranges. The following examples show some ways addressing can be controlled by `sed`. The “-n” option flag suppresses the default behavior of `sed` to display every line of input as a part of the output. In the first example, `sed` operates on lines 4 through 7 from the file. Notice how only the first listed table-row from the file is displayed (lines 4 to 7). Next, `sed` displays only the first and last lines in the file. Some versions of `sed` allow for patterns to specify a range of addresses to apply the command. Notice in the output, commas are only removed from the table, not from the comments.

Example of sed – address ranges:

```
cat << EOF > dummy_table.frag
<!--This, is a comment. -->
<p>This, is a paragraph.</p>
<table border="1">
<tr>
<td>row 1, 1st cell</td>
<td>row 1, 2nd cell</td>
</tr>
<tr>
<td>row 2, 1st cell</td>
<td>row 2, 2nd cell</td>
</tr>
</table>
<!--This, is another comment. -->
EOF

$ sed -n 4,7p dummy_table.frag
<tr>
<td>row 1, 1st cell</td>
<td>row 1, 2nd cell</td>
</tr>

$ sed -n -e 1p -e \$p dummy_table.frag
<!--This, is a comment. -->
```

```
<!--This, is another comment. -->

$ sed '/^<table/,/^<\/table/s/,//g' dummy_table.frag
<!--This, is a comment. -->
<p>This, is a paragraph.</p>
<table border="1">
<tr>
<td>row 1 1st cell</td>
<td>row 1 2nd cell</td>
</tr>
<tr>
<td>row 2 1st cell</td>
<td>row 2 2nd cell</td>
</tr>
</table>
<!--This, is another comment. -->
```

Patterns inside an expression can be grouped and then referenced as part of the output. This can prove useful in a number of contexts such as swapping values or positional variables. The parentheses are used to mark out patterns in the expression, and must be escaped with a backslash \(pattern-here\). The pattern is referenced elsewhere in the expression by using a \n where n is the number of the pattern in order of the marked patterns. Decomposing this expression into components makes it easier to recognize how it works:

Pattern	Comments
`/^#.*$/d`	`remove from output lines starting with a #`
`/^$/d`	`remove from output blank lines`
`s/\([:a-z:]*\):\(.*\) /\2:\1 /`	`This statement marks the first string of lower-case characters ending with a colon, then marks the string of characters following the colon. For output these marked strings swap position.`

Example of sed – grouping patterns:

```
cat << EOF > sed_chown_example.txt
# use sed to swap the group:owner to owner:group

sudo chown dba:jdoe oraenv.ksh
sudo chown staff:jdoe sysenv.ksh
...
EOF

$ sed '/^#.*$/d;/^$/d;s/\([:a-z:]*\):\(.*\) /\2:\1 /' sed_chown_example.txt
sudo chown jdoe:dba oraenv.ksh
sudo chown jdoe:staff sysenv.ksh
...
```

Using awk

The `awk` program can be a handy text manipulator – performing jobs such as parsing, filtering, and easy formatting of text. It takes its input on `stdin` or from files and by default displays output on `stdout`. A variety of releases are available for `awk` under different names like `nawk` and `gawk`. Behavior between versions and vendor releases of `awk` vary. `awk` is different than the other commands reviewed in this article, because it is a programming language. This language provides internal functions for math, string manipulation, flow control, and text formatting. Programmers can

also define their own functions, creating libraries of user-defined functions or stand-alone scripts. Because `awk` contains so many features to demonstrate only a few examples are shown. Please see the Resources section or man pages for more information.

At first, in this example, `awk` is used as a filter to print only full file systems from a Linux system. By default `awk` uses whitespace to identify separate columns. The example examines column 5 since it shows the percentage of disk space used. If the disk utilization is 100%, the first example prints the record on `stdout`. The following statement extends the first to format a message, perhaps to send in an email or write as part of a message to a log file. Then an example is shown on how to create a match using numeric comparison.

Example of awk - filter:

```
$ df –k
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda1             61438632  61381272     57360 100% /
udev                    255788       148    255640   1% /dev
/dev/mapper/datavg     6713132   3584984   3128148  54% /data
rmthost1:/archives/backup    -         -         -   -  /backups
rmthost1:/archives/          -         -         -   -  /amc
rmthost1:/archives/data2     -         -         -   -  /data2

$ df -k |awk '$5 ~ /100%/ {print $0}'
/dev/sda1             61438632  61381272     57360 100% /

$ df -k |awk '$5 ~ /100%/ {printf("full filesystem: %s, mountpoint: %s\n",$6,$1)}'
full filesystem: /, mountpoint: /dev/sda1

$ df -k |awk '$4 > 3000000  {print $0}'
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/mapper/datavg     6713132   3584984   3128148  54% /data
```

Sometimes data is not delimited by whitespace. Take for instance the /etc/passwd file, delimited by the colon “:” character. This example shows how `awk` uses the -F flag to print the username and UID of the first 5 entries listed in /etc/passwd. Next, the `substr()` function of awk is demonstrated by printing the first three characters of column 1 in the /etc/passwd file.

Example of awk - field separator / string function:

```
$ cat /etc/passwd |awk -F: '{printf("%s %s\n", $1,$3)}' |head -5
root 0
daemon 1
bin 2
sys 3
adm 4

cat /etc/passwd |awk -F: '{printf("%s \n", substr($1,1,3))}'|head -5
roo
dae
bin
sys
adm
```

Many times system administrators or programmers write their own `awk` scripts to perform some sort of job. Here is an example of an `awk` program to average numbers found in the third column of a file. The math is done manually by adding the column 3 data up in the total variable. NR is a

special internal variable `awk` uses to keep track of how many records were processed. The average for column 3 is obtained by dividing the total variable by NR. The program displays intermediate results and data so the logic is easier to follow.

Example of awk – program / math:

```
cat << EOF > dummy_file2.dat
011  IBM 174.99
012 INTC 22.78
013 SAP 59.37
014 vmw 102.92
EOF

$ cat avg.awk
awk 'BEGIN {total=0;}
           {printf("tot: %.2f arg3: %.2f NR: %d\n",total, $3, NR); total+=$3;}
     END {printf "Total:%.3f Average:%.3f \n",total,total/NR}'

$ cat dummy_file2.dat | avg.awk
tot: 0.00 arg3: 174.99 NR: 1
tot: 174.99 arg3: 22.78 NR: 2
tot: 197.77 arg3: 59.37 NR: 3
tot: 257.14 arg3: 102.92 NR: 4
Total:360.060 Average:90.015
```

Shell-based string operations

A shell can be a powerful programming language. Similar to awk, shells offer a wide variety of options for string operations, math functionality, arrays, flow control, and file operations. Below are some examples showing how to extract parts of a string from one side. The operation does not change the value of the string, but echoes what the result would look like, and is often used as an assignment to a variable. Use the percent “%” sign to truncate right of the pattern, and use the hash mark “#” to truncate left of the pattern.

Example of shell script – string extraction:

```
$ cat string_example1.sh
#!/bin/sh
FILEPATH=/home/w/wyoes/samples/ksh_samples-v1.0.ksh
echo '${FILEPATH}      =' ${FILEPATH}      "  # the full filepath"
echo '${#FILEPATH}     =' ${#FILEPATH}      "  # length of the string"
echo '${FILEPATH%.*}   =' ${FILEPATH%.*}   "  # truncate right of the last dot"
echo '${FILEPATH%%.*}  =' ${FILEPATH%%.*}  "  # truncate right of the first dot"
echo '${FILEPATH%%/w*} =' ${FILEPATH%%/w*} "  # truncate right of the first /w"
echo '${FILEPATH#/*/*/}  =' ${FILEPATH#/*/*/}  "  # truncate left of the third slash"
echo '${FILEPATH##/*/} =' ${FILEPATH##/*/} "  # truncate left of the last slash"

$ ./string_example1.sh
${FILEPATH}=/home/w/wyoes/samples/ksh_samples-v1.0.ksh # the full filepath
${#FILEPATH} = 42                                      # length of the string
${FILEPATH%.*}=/home/w/wyoes/samples/ksh_samples-v1.0  # truncate right of the last dot
${FILEPATH%%.*}=/home/w/wyoes/samples/ksh_samples-v1   # truncate right of the first dot
${FILEPATH%%/w*}=/home                                 # truncate right of the first /w
${FILEPATH#/*/*/}=wyoes/samples/ksh_samples-v1.0.ksh   # truncate left of the third slash
${FILEPATH##/*/}=ksh_samples-v1.0.ksh                  # truncate left of the last slash
```

As an example, a system administrator may need to change the extension of a set of .jpg files to all lower-case letters. Since UNIX servers are case sensitive, some applications may require the

lower case extension, or maybe the administrator is simply trying to standardize file extensions. Changing a large number of files by hand or through a GUI could take hours. A sample shell script showing a method to solve this problem follows. The example is made up of two files. First is setup_files.ksh, used to set up a sample directory tree, and populate the tree with some files. It also creates a list of files needing extensions changed. The second script called fix_extension.ksh reads in the list of files, changing the extensions where appropriate. As part of the `mv` command, the % string operator is used to truncate to the right of the last dot "." in the filename (truncates the extension). Both scripts also use the `find` command to display what was accomplished after the run.

Example of shell script – change file extension:

```
$ cat setup_files.ksh
mkdir /tmp/mv_demo
[ ! -d /tmp/mv_demo ] && exit

cd /tmp/mv_demo
mkdir tmp JPG 'pictures 1'
touch a.JPG b.jpg c.Jpg d.jPg M.jpG P.jpg JPG_file.JPG JPG.file2.jPg file1.JPG.Jpg 'tmp/
pic 2.Jpg' 10.JPG.bak 'pictures 1/photo.JPG' JPG/readme.txt JPG/sos.JPG

find . -type f|grep -i "\.jpg$" |sort| tee file_list.txt

$ ./setup_files.ksh
./JPG.file2.jPg
./JPG/sos.JPG
./JPG_file.JPG
./M.jpG
./P.jpg
./a.JPG
./b.jpg
./c.Jpg
./d.jPg
./file1.JPG.Jpg
./pictures 1/photo.JPG
./tmp/pic 2.Jpg

$ cd /tmp/mv_demo
$ cat /tmp/fix_extension.ksh
while read f ; do
    mv "${f}" "${f%.*}.jpg"
done < file_list.txt

find . -type f|grep -i "\.jpg$" |sort

$ /tmp/fix_extension.ksh
./JPG.file2.jpg
./JPG/sos.jpg
./JPG_file.jpg
./M.jpg
./P.jpg
./a.jpg
./b.jpg
./c.jpg
./d.jpg
./file1.JPG.jpg
./pictures 1/photo.jpg
./tmp/pic 2.jpg
```

In the spirit of creating useful and reusable tools, the example for changing file extensions should be more generalized. Some improvements that come to mind are to pass in a stream of filenames

to be changed, such as part of a pipeline. Option flags could be added to specify file extensions (like .mp3 or .mov) to change, and how to format the file extension such as lower, upper, or mixed case. The possibilities are limited only by the programmer’s imagination and time.

Summary

UNIX offers a wide variety of tools to do text parsing natively, in many cases without the need to rely on special interpreters which may not be installed. This article is rather broad in its survey of commands when compared to depth of their usage. The commands in this article are partially demonstrated, some systems implement flags or behavior differently than other systems. UNIX certainly offers more commands and ways of accomplishing the same tasks – “There's more than one way to do it”.

Resources

Learn

- See Advanced Bash-Scripting Guide - Manipulating Strings to learn more about bash scripting
- Database join explained
- Go to the sed $HOME page for information.
- Learn more about Sed.
- Learn how to use the UNIX stream editor, sed.
- Gawk: Effective AWK Programming: Read the standard user manual for GAWK.
- Learn more with this tutorial: Awk - An Introduction and Tutorial by Bruce Barnett
- Read an intro to awk
- The Open Group Base Specifications Issue 7 (awk)
- The Art of UNIX Programming

Get products and technologies

- Go to the KornShell home page for software and information.
- Try out IBM software for free. Download a trial version, log into an online trial, work with a product in a sandbox environment, or access it through the cloud. Choose from over 100 IBM product trials.

Discuss

- Follow developerWorks on Twitter.
- Participate in developerWorks blogs and get involved in the developerWorks community.
- Get involved in the My developerWorks community.
- Participate in the AIX and UNIX® forums:
  - AIX Forum
  - AIX Forum for developers
  - Cluster Systems Management
  - Performance Tools Forum
  - Virtualization Forum
  - More AIX and UNIX Forums

About the author

Brad Yoes

Brad Yoes is a Migration Engineer with IBM, with 15 years experience migrating applications, who began working with UNIX in 1993. Outside of work he enjoys being with his family, reading non-fiction books, and jogging.