

Quick Guide to sed and gawk

IN THIS APPENDIX

The basics for using sed

What you need to know about gawk

If you do any type of data handling in your shell scripts, most likely you'll need to use either the sed program or the gawk program (and sometimes both). This appendix provides a quick reference for sed and gawk commands that come in handy when working with data in your shell scripts.

The sed Editor

The sed editor can manipulate data in a data stream based on commands you either enter into the command line or store in a command text file. It reads one line of data at a time from the input and matches that data with the supplied editor commands, changes data in the stream as specified in the commands, and then outputs the new data to STDOUT.

Starting the sed editor

Here's the format for using the sed command:

```
sed options script file
```

The *options* parameters allow you to customize the behavior of the sed command and include the options shown in Table B-1.

TABLE B-1 The sed Command Options

Option	Description
-e <i>script</i>	Adds commands specified in <i>script</i> to the commands run while processing the input
-f <i>file</i>	Adds the commands specified in the file <i>file</i> to the commands run while processing the input
-n	Doesn't produce output for each command, but waits for the print command

The *script* parameter specifies a single command to apply against the stream data. If more than one command is required, you must use either the `-e` option to specify them in the command line or the `-f` option to specify them in a separate file.

sed commands

The *sed* editor script contains commands that *sed* processes for each line of data in the input stream. This section describes some of the more common *sed* commands you'll want to use.

Substitution

The *s* command substitutes text in the input stream. Here's the format of the *s* command:

s/pattern/replacement/flags

pattern is the text to replace, and *replacement* is the new text that *sed* inserts in its place.

The *flags* parameter controls how the substitution takes place. Four types of substitution flags are available:

- A number indicates the pattern occurrence that should be replaced.
- *g* indicates that all occurrences of the text should be replaced.
- *p* indicates that the contents of the original line should be printed.
- *w file* indicates that the results of the substitution should be written to a file.

In the first type of substitution, you can specify which occurrence of the matching pattern the *sed* editor should replace. For example, you use the number 2 to replace only the second occurrence of the pattern.

Addressing

By default, the commands you use in the *sed* editor apply to all lines of the text data. If you want to apply a command to only a specific line, or a group of lines, you must use *line addressing*.

There are two forms of line addressing in the *sed* editor:

- A numeric range of lines
- A text pattern that filters out a line

Both forms use the same format for specifying the address:

[address]command

When using numeric line addressing, you reference lines by their line position in the text stream. The `sed` editor assigns the first line in the text stream as line number 1 and continues sequentially for each new line.

```
$ sed '2,3s/dog/cat/' data1
```

The other method of restricting which lines a command applies to is a bit more complicated. The `sed` editor allows you to specify a text pattern that it uses to filter lines for the command. Here's the format for this:

```
/pattern/command
```

You must encapsulate the *pattern* you specify in forward slashes. The `sed` editor applies the command only to lines that contain the text pattern that you specify.

```
$ sed '/rich/s/bash/csh/' /etc/passwd
```

This filter finds the line that contains the text `rich` and replaces the text `bash` with `csh`.

You can also group more than one command together for a specific address:

```
address {
    command1
    command2
    command3 }
```

The `sed` editor applies each of the commands you specify only to lines that match the address specified. The `sed` editor processes each command listed on the address line(s):

```
$ sed '2{
> s/fox/elephant/
> s/dog/cat/
> }' data1
```

The `sed` editor applies each of the substitutions to the second line in the data file.

Deleting lines

The delete command, `d`, pretty much does what it says. It deletes any text lines that match the addressing scheme supplied. Be careful with the delete command, because if you forget to include an addressing scheme, all the lines are deleted from the stream:

```
$ sed 'd' data1
```

The delete command is obviously most useful when used in conjunction with a specified address. This allows you to delete specific lines of text from the data stream, either by line number:

```
$ sed '3d' data6
```

or by a specific range of lines:

```
$ sed '2,3d' data6
```

The pattern-matching feature of the `sed` editor also applies to the delete command:

```
$ sed '/number 1/d' data6
```

Only lines matching the specified text are deleted from the stream.

Inserting and appending text

As you would expect, like any other editor, the `sed` editor allows you to insert and append text lines to the data stream. The difference between the two actions can be confusing:

- The insert command (`i`) adds a new line before the specified line.
- The append command (`a`) adds a new line after the specified line.

The format of these two commands can be confusing: You can't use these commands on a single command line. You must specify the line to insert or append on a separate line by itself. Here's the format for doing this:

```
sed '[address]command\  
new line'
```

The text in `new line` appears in the `sed` editor output in the place you specify. Remember that when you use the insert command, the text appears before the data stream text:

```
$ echo "testing" | sed 'i\  
> This is a test'  
This is a test  
testing  
$
```

And when you use the append command, the text appears after the data stream text:

```
$ echo "testing" | sed 'a\  
> This is a test'  
testing  
This is a test  
$
```

This allows you to insert text at the end of the normal text.

Changing lines

The change command allows you to change the contents of an entire line of text in the data stream. It works the same as the insert and append commands, in that you must specify the new line separately from the rest of the `sed` command:

```
$ sed '3c\  
> This is a changed line of text.' data6
```

The backslash character is used to indicate the new line of data in the script.

Transform command

The transform command (*y*) is the only *sed* editor command that operates on a single character. The transform command uses this format:

```
[address]y/inchars/outchars/
```

The transform command performs a one-to-one mapping of the *inchars* and the *outchars* values. The first character in *inchars* is converted to the first character in *outchars*. The second character in *inchars* is converted to the second character in *outchars*. This mapping continues throughout the length of the specified characters. If the *inchars* and *outchars* are not the same length, the *sed* editor produces an error message.

Printing lines

Similar to the *p* flag in the substitution command, the *p* command prints a line in the *sed* editor output. The most common use for the print command is for printing lines that contain matching text from a text pattern:

```
$ sed -n '/number 3/p' data6
This is line number 3.
$
```

The print command allows you to filter only specific lines of data from the input stream.

Writing to a file

The *w* command is used to write lines to a file. Here's the format for the *w* command:

```
[address]w filename
```

The *filename* can be specified as either a relative or absolute pathname, but in either case, the person running the *sed* editor must have write permissions for the file. The *address* can be any type of addressing method used in *sed*, such as a single line number, a text pattern, or a range of line numbers or text patterns.

Here's an example that prints only the first two lines of a data stream to a text file:

```
$ sed '1,2w test' data6
```

The output file *test* contains only the first two lines from the input stream.

Reading from a file

You've already seen how to insert and append text into a data stream from the *sed* command line. The read command (*r*) allows you to insert data contained in a separate file.

Here's the format of the read command:

```
[address]r filename
```

The *filename* parameter specifies either an absolute or relative pathname for the file that contains the data. You can't use a range of addresses for the read command. You can specify only a single line number or text pattern address. The *sed* editor inserts the text from the file after the address.

```
$ sed '3r data' data2
```

The *sed* editor inserts the complete text from the *data* file into the *data2* file, starting at line 3 of the *data2* file.

The gawk Program

The *gawk* program is the GNU version of the original *awk* program in Unix. The *awk* program takes stream editing one step further than the *sed* editor by providing a programming language instead of just editor commands. This section describes the basics of the *gawk* program as a quick reference to its abilities.

The gawk command format

The basic format of the *gawk* program is as follows:

```
gawk options program file
```

Table B-2 shows the options available with the *gawk* program.

TABLE B-2 The gawk Options

Option	Description
-F <i>fs</i>	Specifies a file separator for delineating data fields in a line
-f <i>file</i>	Specifies a file name to read the program from
-v <i>var=value</i>	Defines a variable and default value used in the <i>gawk</i> program
-mF <i>N</i>	Specifies the maximum number of fields to process in the data file
-mr <i>N</i>	Specifies the maximum record size in the data file
-W <i>keyword</i>	Specifies the compatibility mode or warning level for <i>gawk</i> . Use the <i>help</i> option to list all the available keywords.

The command line options provide an easy way to customize features in the *gawk* program.

Using gawk

You can use `gawk` either directly from the command line or from within your shell scripts. This section demonstrates how to use the `gawk` program and how to enter scripts for `gawk` to process.

Reading the program script from the command line

A `gawk` program script is defined by an opening and closing brace. You must place script commands between the two braces. Because the `gawk` command line assumes that the script is a single text string, you must also enclose your script in single quotation marks. Here's an example of a simple `gawk` program script specified on the command line:

```
$ gawk '{print $1}'
```

This script displays the first data field in every line of the input stream.

Using multiple commands in the program script

A programming language wouldn't be very useful if you could execute only one command. The `gawk` programming language allows you to combine commands into a normal program. To use multiple commands in the program script specified on the command line, just place a semicolon between commands:

```
$ echo "My name is Rich" | gawk '{$4="Dave"; print $0}'
My name is Dave
$
```

The script performs two commands: It replaces the fourth data field with a different value, and then it displays the entire data line in the stream.

Reading the program from a file

As with the `sed` editor, the `gawk` editor allows you to store your programs in a file and refer to them in the command line:

```
$ cat script2
{ print $5 "'s userid is " $1 }
$ gawk -F: -f script2 /etc/passwd
```

The `gawk` program processes all the commands specified in the file on the input stream data.

Running scripts before processing data

The `gawk` program also allows you to specify when the program script is run. By default, `gawk` reads a line of text from the input and then executes the program script on the data in the line of text. Sometimes, you may need to run a script before processing data, such as to create a header section for a report. To do that, you use the `BEGIN` keyword. This forces

gawk to execute the program script specified after the `BEGIN` keyword before reading the data:

```
$ gawk 'BEGIN {print "This is a test report"}'
This is a test report
$
```

You can place any type of gawk command in the `BEGIN` section, such as commands that assign default values to variables.

Running scripts after processing data

Similar to the `BEGIN` keyword, the `END` keyword allows you to specify a program script that gawk executes after reading the data:

```
$ gawk 'BEGIN {print "Hello World!"} {print $0} END {print
    "byebye"}' data1
Hello World!
This is a test
This is a test
This is another test.
This is another test.
byebye
$
```

The gawk program executes the code in the `BEGIN` section first, then processes any data in the input stream, and then executes the code in the `END` section.

The gawk variables

The gawk program is more than just an editor; it's a complete programming environment. As such, lots of commands and features are associated with gawk. This section shows the main features you need to know for programming with gawk.

Built-in variables

The gawk program uses built-in variables to reference specific features within the program data. This section describes the gawk built-in variables available for you to use in your gawk programs and demonstrates how to use them.

The gawk program defines data as records and data fields. A *record* is a line of data (delineated by the newline characters by default), and a *data field* is a separate data element within the line (delineated by a white space character, such as a space or tab, by default).

The gawk program uses data field variables to reference data elements within each record. Table B-3 describes these variables.

TABLE B-3 The gawk Data Field and Record Variables

Variable	Description
\$0	The entire data record
\$1	The first data field in the record
\$2	The second data field in the record
\$n	The nth data field in the record
FIELDWIDTHS	A space-separated list of numbers defining the exact width (in spaces) of each data field
FS	Input field separator character
RS	Input record separator character
OFS	Output field separator character
ORS	Output record separator character

In addition to the field and record separator variables, gawk provides some other built-in variables to help you know what’s going on with your data and extract information from the shell environment. Table B-4 shows the other built-in variables in gawk.

TABLE B-4 More gawk Built-In Variables

Variable	Description
ARGC	The number of command line parameters present
ARGIND	The index in ARGV of the current file being processed
ARGV	An array of command line parameters
CONVFMT	The conversion format for numbers (see the <code>printf</code> statement), with a default value of <code>%.6 g</code>
ENVIRON	An associative array of the current shell environment variables and their values
ERRNO	The system error if an error occurs reading or closing input files
FILENAME	The filename of the data file used for input to the gawk program
FNR	The current record number in the data file
IGNORECASE	If set to a non-zero value, gawk all string functions (including regular expressions); ignore the case of characters.
NF	The total number of data fields in the data file
NR	The number of input records processed

Continues

TABLE B-4 (continued)

Variable	Description
OFMT	The output format for displaying numbers, with a default of %.6 g
RLENGTH	The length of the substring matched in the match function
RSTART	The start index of the substring matched in the match function

You can use the built-in variables anywhere in the gawk program script, including the BEGIN and END sections.

Assigning variables in scripts

Assigning values to variables in gawk programs is similar to how you assign values to variables in a shell script — using an *assignment statement*:

```
$ gawk '  
> BEGIN{  
>   testing="This is a test"  
>   print testing  
> }'  
This is a test  
$
```

After you assign a value to a variable, you can use that variable anywhere in your gawk script.

Assigning variables in the command line

You can also use the gawk command line to assign values to variables for the gawk program. This allows you to set values outside of the normal code, changing values on the fly. Here's an example of using a command line variable to display a specific data field in the file:

```
$ cat script1  
BEGIN{FS=","}  
{print $n}  
$ gawk -f script1 n=2 data1  
$ gawk -f script1 n=3 data1
```

This feature is a great way to process data from your shell scripts in the gawk script.

The gawk program features

Some features of the gawk program make it handy for manipulating data, allowing you to create gawk scripts that can parse just about any type of text file, including log files.

Regular expressions

You can use either a Basic Regular Expression (BRE) or an Extended Regular Expression (ERE) to filter the lines in the data stream to which the program script applies.

When using a regular expression, the regular expression must appear before the left brace of the program script that it controls:

```
$ gawk 'BEGIN{FS=","} /test/{print $1}' data1
This is a test
$
```

The matching operator

The *matching operator* allows you to restrict a regular expression to a specific data field in the records. The matching operator is the tilde character (~). You specify the matching operator, along with the data field variable, and the regular expression to match:

```
$1 ~ /^data/
```

This expression filters records where the first data field starts with the text data.

Mathematical expressions

In addition to regular expressions, you can also use mathematical expressions in the matching pattern. This feature comes in handy when matching numerical values in data fields. For example, if you want to display all the system users who belong to the root users group (group number 0), you could use this script:

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
```

This script displays the first data field value for all lines that contain the value 0 in the fourth data field.

Structured commands

The gawk program supports the structured commands discussed in this section.

The *if-then-else* statement:

```
if (condition) statement1; else statement2
```

The *while* statement:

```
while (condition)
{
    statements
}
```

The do-while statement:

```
do {  
    statements  
} while (condition)
```

The for statement:

```
for(variable assignment; condition; iteration process)
```

This provides a wealth of programming opportunities for the gawk script programmer. You can write gawk programs that rival the functions of just about any higher-level language program.