Unit IV

What Are Regular Expressions?

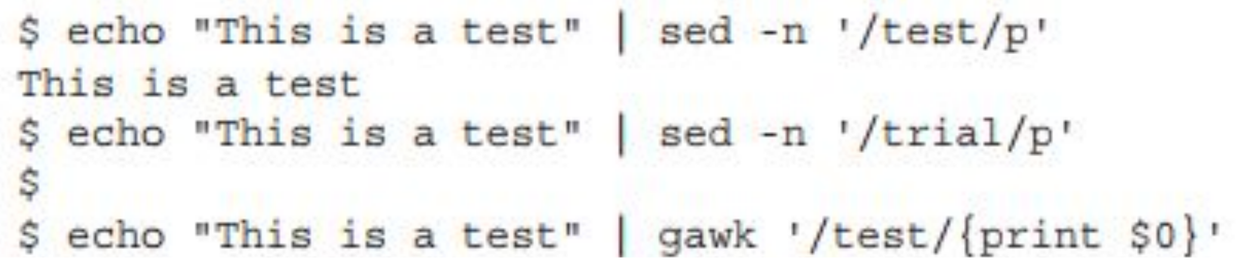
- A *regular expression* is a pattern template you define that a Linux utility uses to filter text. A Linux utility (such as the `sed` editor or the `gawk` program) matches the regular expression pattern against data as that data flows into the utility. If the data matches the pattern, it's accepted for processing. If the data doesn't match the pattern, it's rejected.
  The Linux world has two popular regular expression engines:
  ■ The POSIX Basic Regular Expression (BRE) engine

  ■ The POSIX Extended Regular Expression (ERE) engine

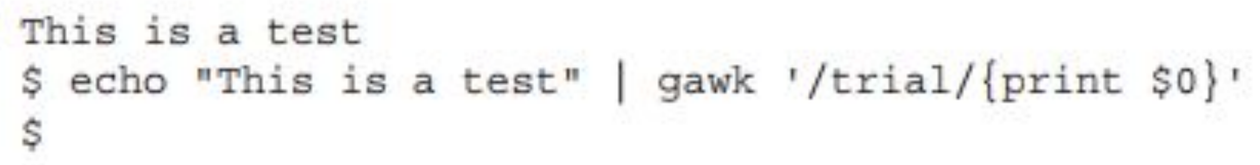
- Most Linux utilities at a minimum conform to the POSIX BRE engine specifications**, recognizing all the pattern symbols it defines. Unfortunately, some utilities (such as the `sed` editor) conform only to a subset of the BRE engine specifications**. This is due to speed constraints, because the `sed` editor attempts to process text in the data stream as quickly as possible.

- The POSIX ERE engine is often found in programming languages that rely on regular expressions **for text filtering. It provides advanced pattern symbols as well as special symbols for common patterns, such as matching digits, words, and alphanumeric characters.** The `gawk` program uses the ERE engine to process its regular expression patterns.

- **Defining BRE Patterns:**
- **Plain text :**
  This Chapter demonstrated how to use standard text strings in the `sed` editor and the `gawk` program to filter data. Here’s an example to refresh your memory:



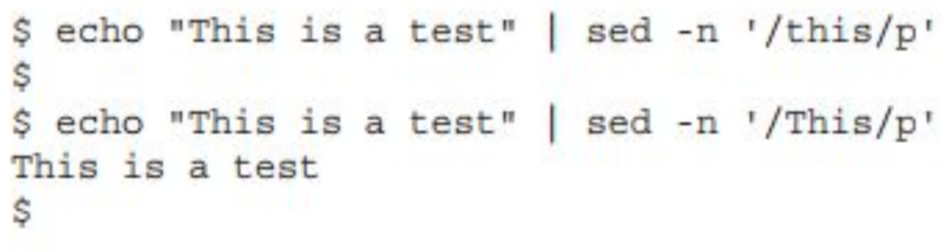
```
$ echo "This is a test" | sed -n '/test/p'
This is a test
$ echo "This is a test" | sed -n '/trial/p'
$
$ echo "This is a test" | gawk '/test/{print $0}'
```



```
This is a test
$ echo "This is a test" | gawk '/trial/{print $0}'
$
```

- The first pattern defines a single word, *test*. The `sed` editor and `gawk` program scripts each use their own version of the `print` command to print any lines that match the regular expression pattern. Because the `echo` statement contains the word “test” in the text string, the data stream text matches the defined regular expression pattern, and the sed editor displays the line.
- The second pattern again defines just a single word, this time the word “trial.” Because the `echo` statement text string doesn’t contain that word, the regular expression pattern doesn’t match, so neither the `sed` editor nor the gawk program prints the line.

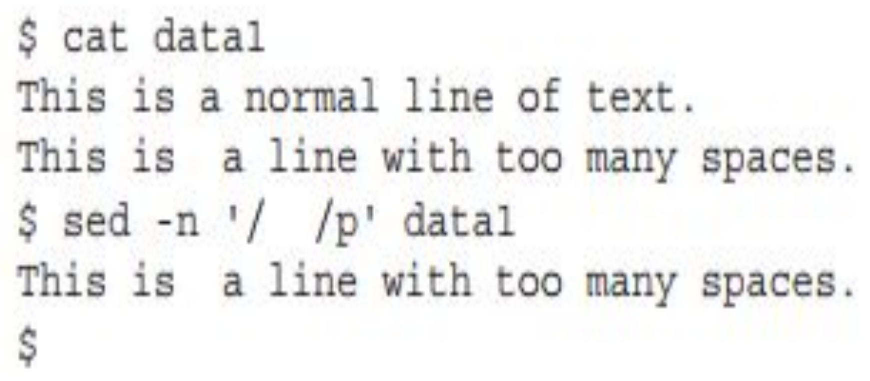
- The first rule to remember is that regular expression patterns are case sensitive. This means they'll match only those patterns with the **proper case of characters**:



```
$ echo "This is a test" | sed -n '/this/p'
$
$ echo "This is a test" | sed -n '/This/p'
This is a test
$
```

- The first attempt failed to match because the word "this" doesn't appear in all lowercase in the text string, while the second attempt, which uses the uppercase letter in the pattern, worked just fine.

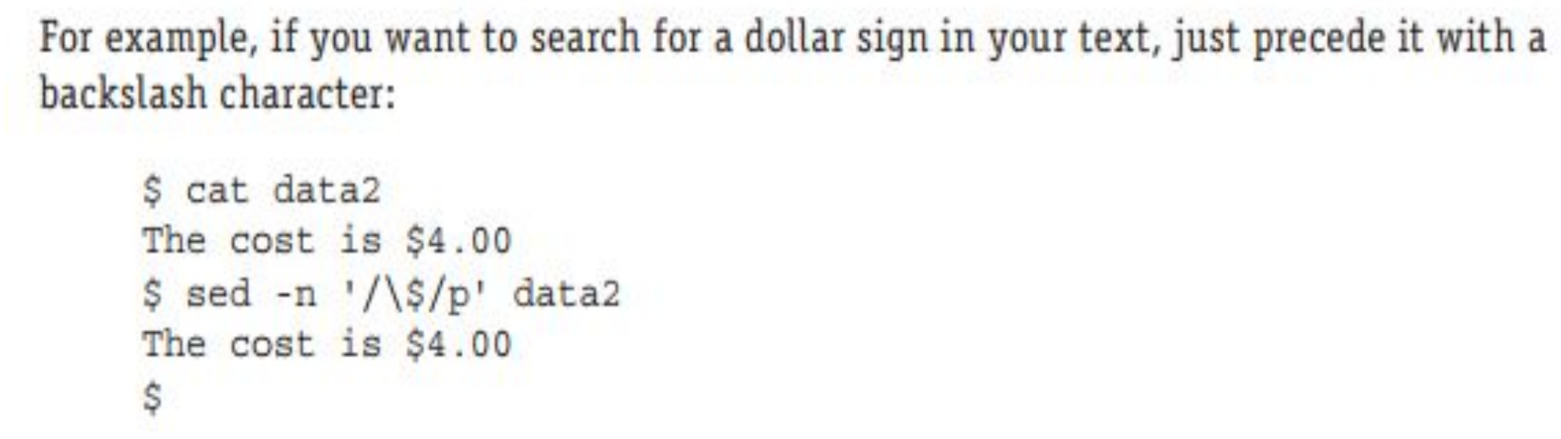
- If you define a space in the regular expression, it must appear in the data stream. You can even create a regular expression pattern that matches multiple contiguous spaces:

- 

```
$ cat data1
This is a normal line of text.
This is  a line with too many spaces.
$ sed -n '/  /p' data1
This is  a line with too many spaces.
$
```

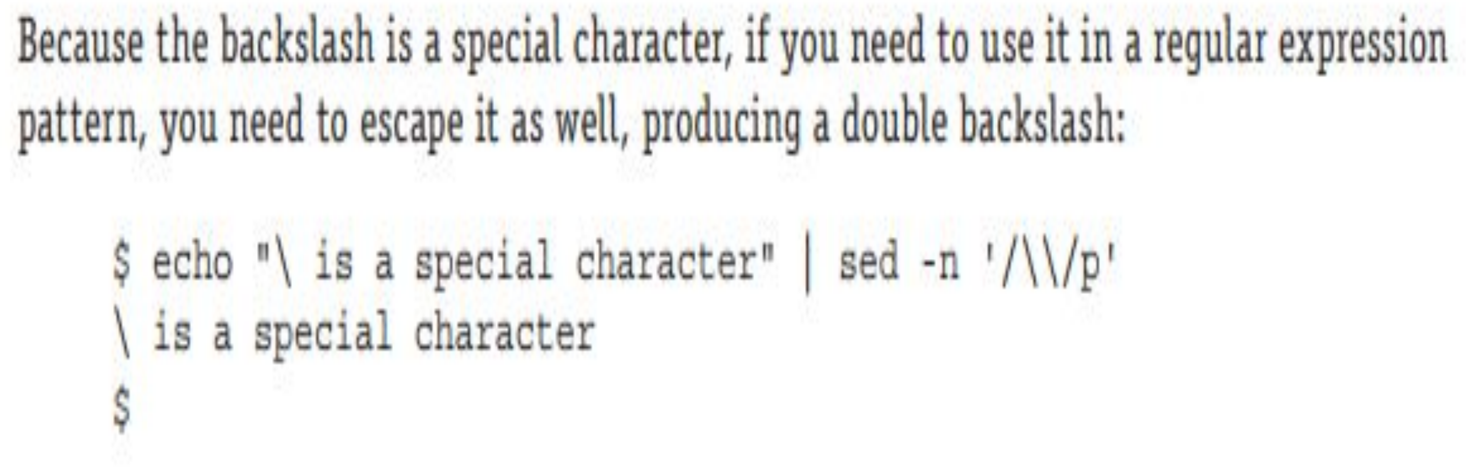
- **Special characters :**Regular expression patterns assign a special meaning to a few characters. If you try to
  use these characters in your text pattern, you won’t get the results you were expecting.
  These special characters are recognized by regular expressions:

```
.*[]^${}\+?|()
```



For example, if you want to search for a dollar sign in your text, just precede it with a backslash character:

```
$ cat data2
The cost is $4.00
$ sed -n '/\$/p' data2
The cost is $4.00
$
```

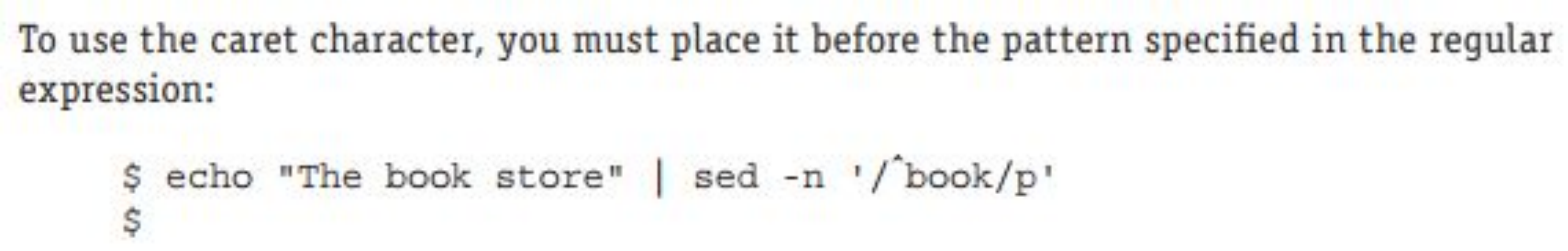


Because the backslash is a special character, if you need to use it in a regular expression pattern, you need to escape it as well, producing a double backslash:

```
$ echo "\ is a special character" | sed -n '/\\/p'
\ is a special character
$
```

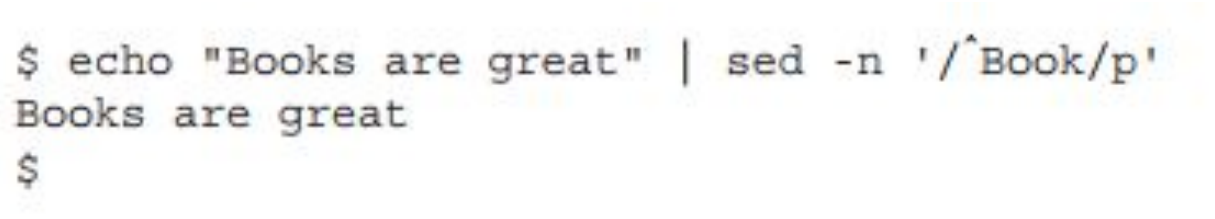
- **Anchor characters:**

  Starting at the beginning The caret character (^) defines a pattern that **starts at the beginning of a line of text in the data stream.** If the pattern is located any place other than the start of the line of text,
  the regular expression pattern fails.



To use the caret character, you must place it before the pattern specified in the regular expression:

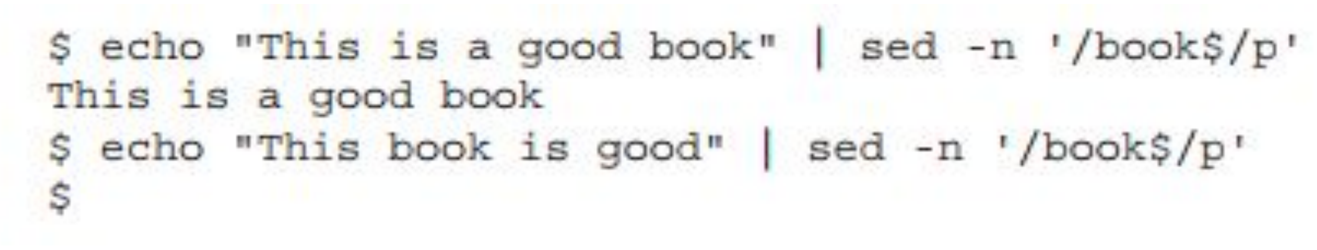
```
$ echo "The book store" | sed -n '/^book/p'
$
```



```
$ echo "Books are great" | sed -n '/^Book/p'
Books are great
$
```

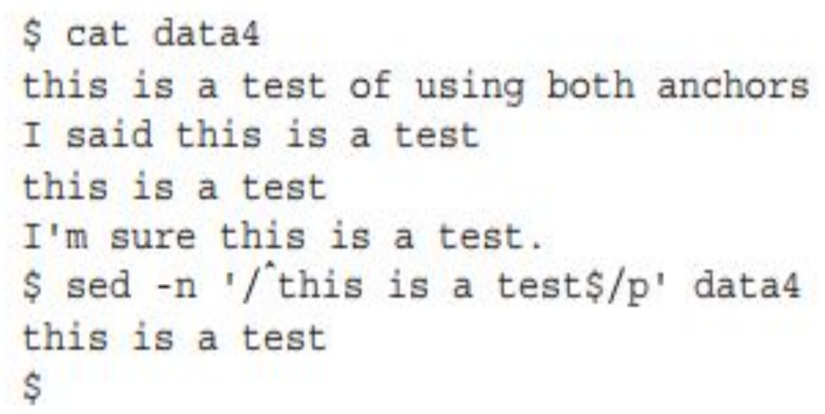
• Looking for the ending:

- The opposite of looking for a pattern at the start of a line is **looking for it at the end of a line**. The dollar sign ($) special character defines the end anchor. Add this special character after a text pattern to indicate that the line of data must end with the text pattern:



```
$ echo "This is a good book" | sed -n '/book$/p'
This is a good book
$ echo "This book is good" | sed -n '/book$/p'
$
```

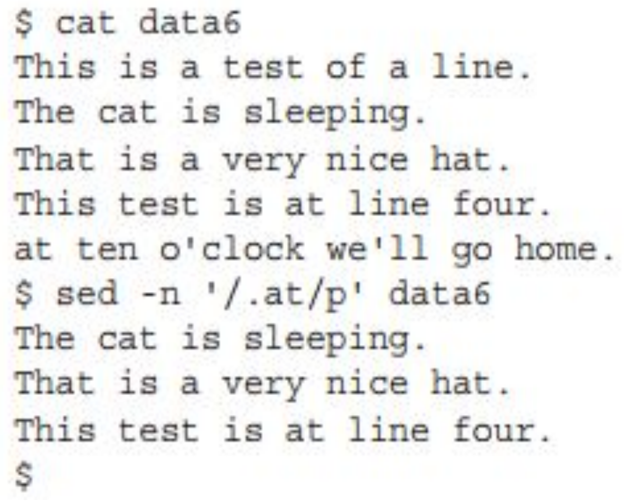
- **Combining anchors**
  In some common situations, you can combine both the start and end anchor on the same line. In the first situation, suppose you want to look for a line of data containing only a specific text pattern:



```
$ cat data4
this is a test of using both anchors
I said this is a test
this is a test
I'm sure this is a test.
$ sed -n '/^this is a test$/p' data4
this is a test
$
```

• The dot character

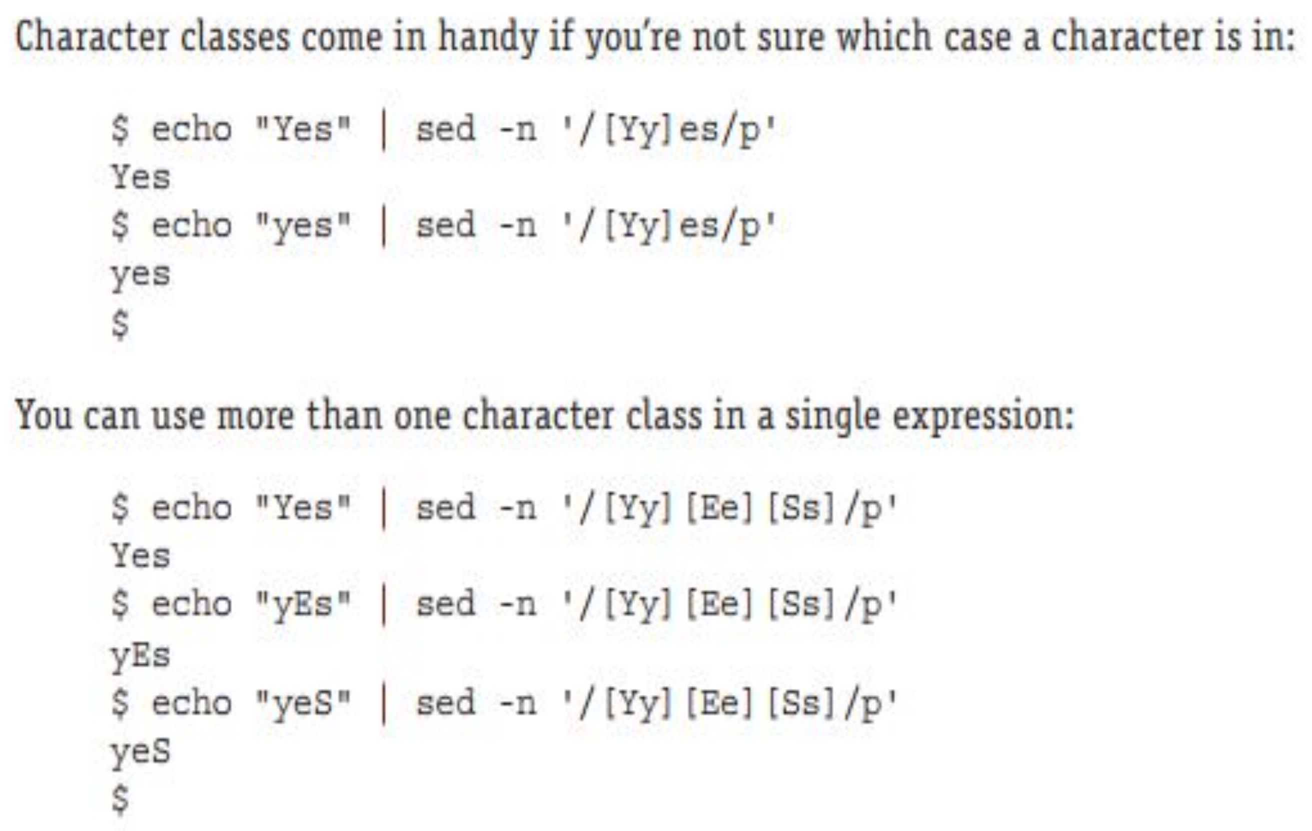
- The dot special character is used to match any single character except a newline character. The dot character must match a character, however; if there's no character in the place of the dot, then the pattern fails.



```
$ cat data6
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.
$ sed -n '/.at/p' data6
The cat is sleeping.
That is a very nice hat.
This test is at line four.
$
```

- you should be able to figure out why the first line failed and why the second and third lines passed. The fourth line is a little tricky. Notice that we matched the `at`, but there's no character in front of it to match the dot character. Ah, but there is! In regular expressions, spaces count as characters, so the space in front of the `at` matches the pattern.

- **Character classes :**
  You can define a class of characters that would match a position in a text pattern. If one of the characters from the character class is in the data stream, it matches the pattern. To define a character class, you use square brackets. The brackets should contain any character you want to include in the class. You then use the entire class within a pattern just like any other wildcard character.

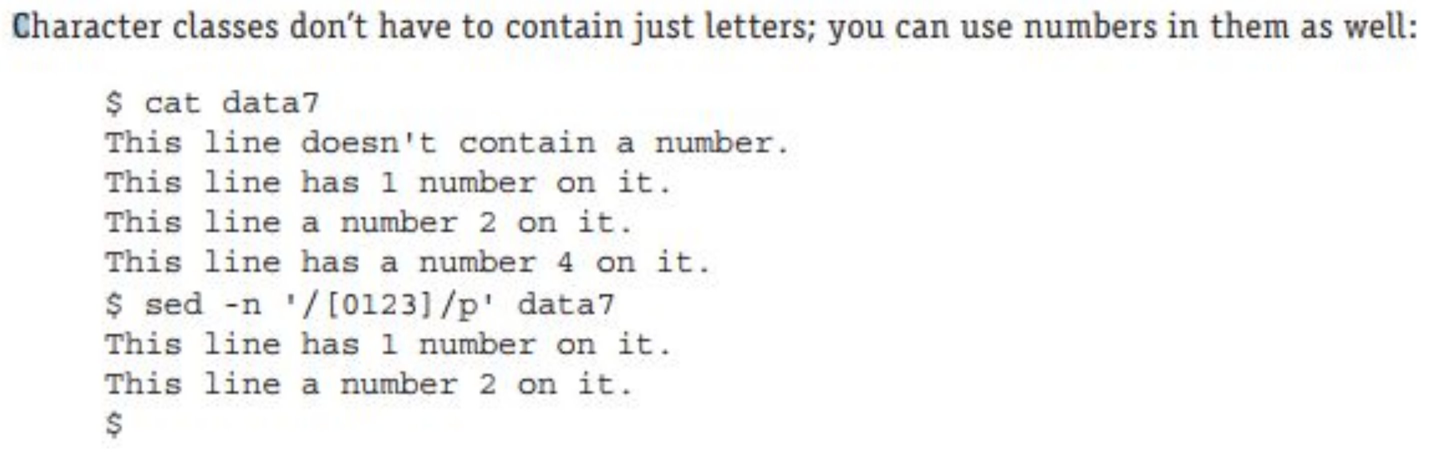


Character classes come in handy if you're not sure which case a character is in:

```
$ echo "Yes" | sed -n '/[Yy]es/p'
Yes
$ echo "yes" | sed -n '/[Yy]es/p'
yes
$
```

You can use more than one character class in a single expression:

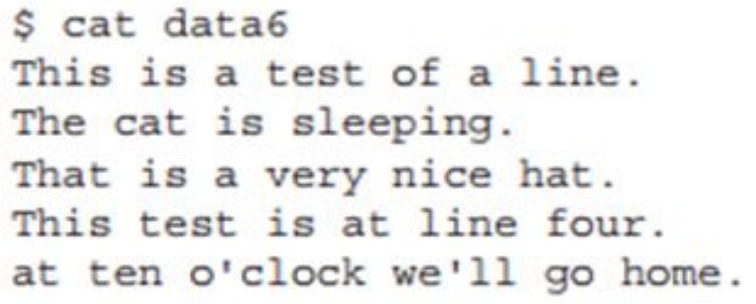
```
$ echo "Yes" | sed -n '/[Yy][Ee][Ss]/p'
Yes
$ echo "yEs" | sed -n '/[Yy][Ee][Ss]/p'
yEs
$ echo "yeS" | sed -n '/[Yy][Ee][Ss]/p'
yeS
$
```


Character classes don't have to contain just letters; you can use numbers in them as well:

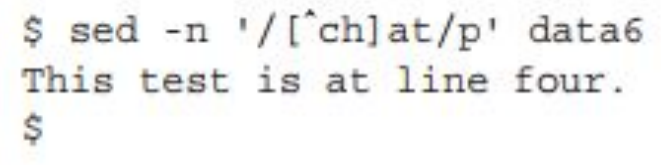
```
$ cat data7
This line doesn't contain a number.
This line has 1 number on it.
This line a number 2 on it.
This line has a number 4 on it.
$ sed -n '/[0123]/p' data7
This line has 1 number on it.
This line a number 2 on it.
$
```

The regular expression pattern matches any lines that contain the numbers 0, 1, 2, or 3. Any other numbers are ignored, as are lines without numbers in them.

Negating character classes



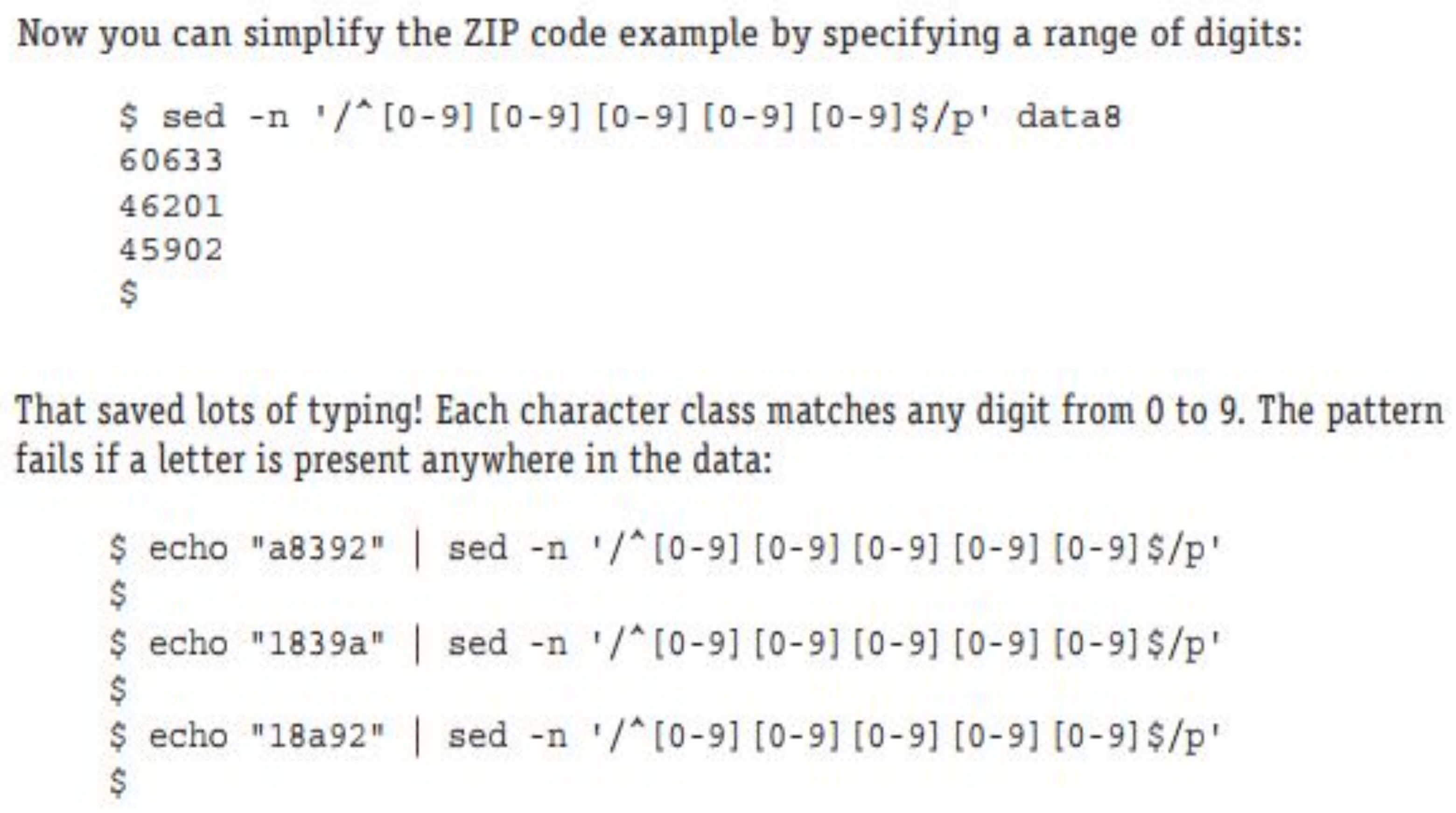
```
$ cat data6
This is a test of a line.
The cat is sleeping.
That is a very nice hat.
This test is at line four.
at ten o'clock we'll go home.
```



```
$ sed -n '/[^ch]at/p' data6
This test is at line four.
$
```

By negating the character class, the regular expression pattern matches any character that's neither a *c* nor an *h*, along with the text pattern. Because the space character fits this category, it passed the pattern match. However, even with the negation, the character class must still match a character, so the line with the `at` in the start of the line still doesn't match the pattern.

- You can use a range of characters within a character class by using the dash symbol. Just specify the first character in the range, a dash, and then the last character in the range. The regular expression includes any character that’s within the specified character range, according to the character set used by the Linux system



Now you can simplify the ZIP code example by specifying a range of digits:

```
$ sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p' data8
60633
46201
45902
$
```

That saved lots of typing! Each character class matches any digit from 0 to 9. The pattern fails if a letter is present anywhere in the data:

```
$ echo "a8392" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
$ echo "1839a" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
$ echo "18a92" | sed -n '/^[0-9][0-9][0-9][0-9][0-9]$/p'
$
```

- **Special character classes**

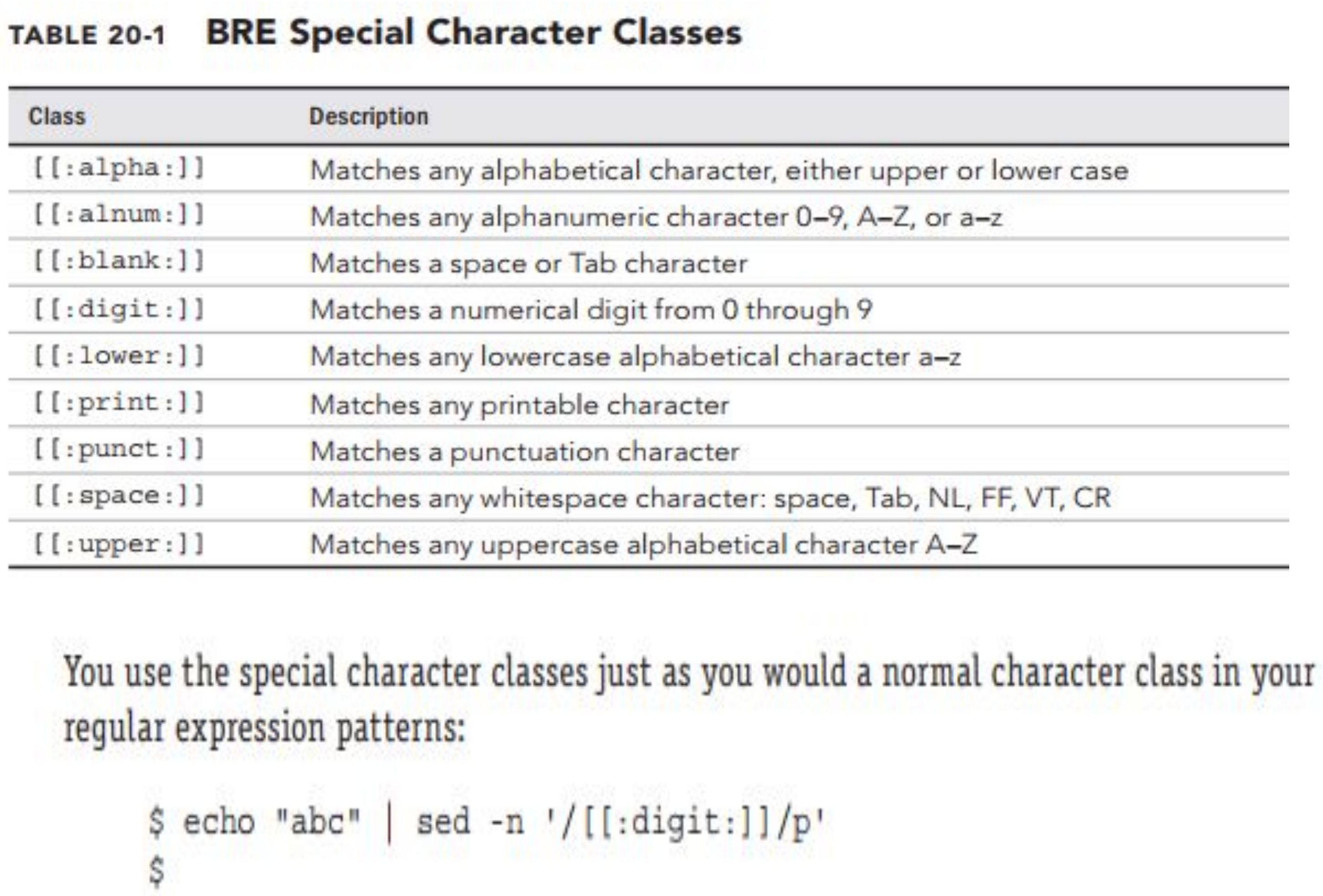


TABLE 20-1 **BRE Special Character Classes**

Class	Description
`[[:alpha:]]`	Matches any alphabetical character, either upper or lower case
`[[:alnum:]]`	Matches any alphanumeric character 0–9, A–Z, or a–z
`[[:blank:]]`	Matches a space or Tab character
`[[:digit:]]`	Matches a numerical digit from 0 through 9
`[[:lower:]]`	Matches any lowercase alphabetical character a–z
`[[:print:]]`	Matches any printable character
`[[:punct:]]`	Matches a punctuation character
`[[:space:]]`	Matches any whitespace character: space, Tab, NL, FF, VT, CR
`[[:upper:]]`	Matches any uppercase alphabetical character A–Z

You use the special character classes just as you would a normal character class in your regular expression patterns:

```
$ echo "abc" | sed -n '/[[:digit:]]/p'
$
```

```
$ echo "abc" | sed -n '/[[:alpha:]]/p'
abc
$ echo "abc123" | sed -n '/[[:digit:]]/p'
abc123
$ echo "This is, a test" | sed -n '/[[:punct:]]/p'
This is, a test
$ echo "This is a test" | sed -n '/[[:punct:]]/p'
$
```

The asterisk

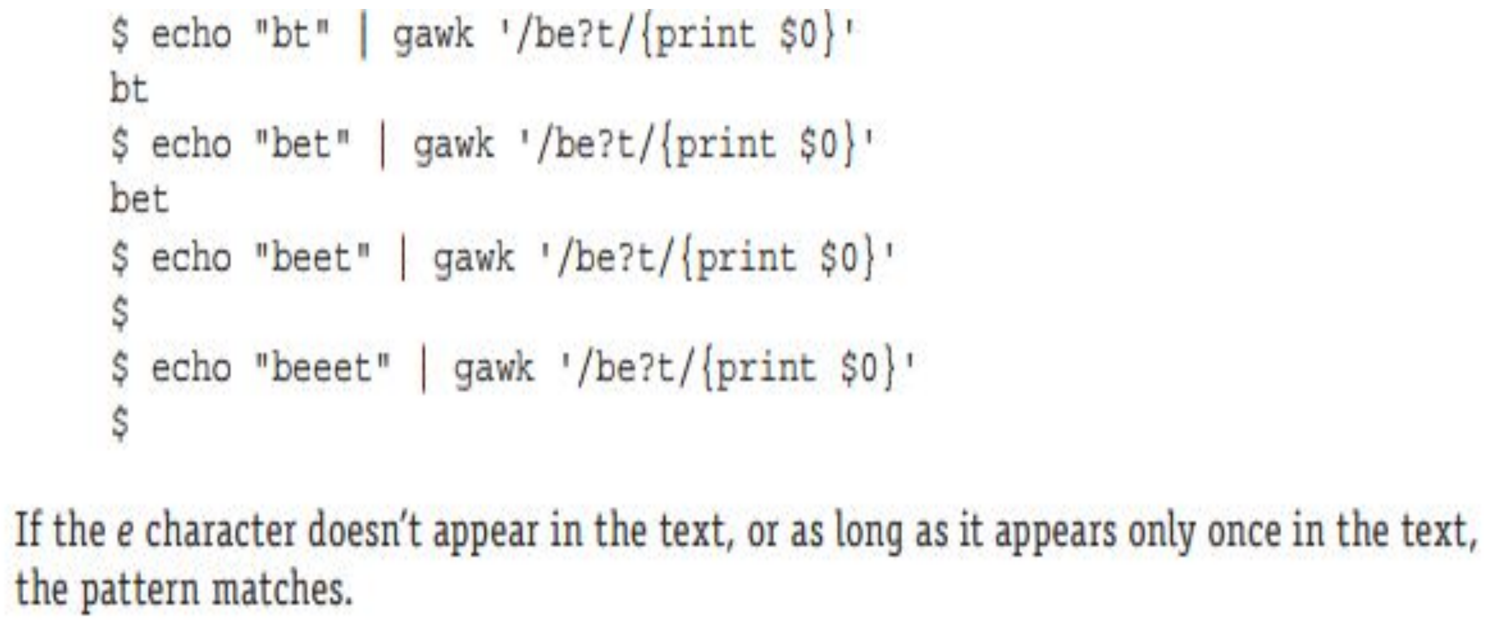
Placing an asterisk after a character signifies that the character must appear zero or more times in the text to match the pattern:

```
$ echo "ik" | sed -n '/ie*k/p'
ik
$ echo "iek" | sed -n '/ie*k/p'
iek
$ echo "ieek" | sed -n '/ie*k/p'
ieek
$ echo "ieeek" | sed -n '/ie*k/p'
ieeek
$ echo "ieeeek" | sed -n '/ie*k/p'
ieeeek
$
```

- The asterisk can also be applied to a character class. This allows you to specify a group or range of characters that can appear more than once in the text:

```
$ echo "bt" | sed -n '/b[ae]*t/p'
bt
$ echo "bat" | sed -n '/b[ae]*t/p'
bat
$ echo "bet" | sed -n '/b[ae]*t/p'
bet
$ echo "btt" | sed -n '/b[ae]*t/p'
btt
$
$ echo "baat" | sed -n '/b[ae]*t/p'
baat
$ echo "baaeeet" | sed -n '/b[ae]*t/p'
baaeeet
$ echo "baeeaeeat" | sed -n '/b[ae]*t/p'
baeeaeeat

$ echo "baakeeet" | sed -n '/b[ae]*t/p'
```

- **Extended Regular Expressions:**
- The question mark is similar to the asterisk, but with a slight twist. The question mark indicates that the **preceding character can appear zero or one time**, but that’s all. It doesn’t match repeating occurrences of the character:



```
$ echo "bt" | gawk '/be?t/{print $0}'
bt
$ echo "bet" | gawk '/be?t/{print $0}'
bet
$ echo "beet" | gawk '/be?t/{print $0}'
$
$ echo "beeet" | gawk '/be?t/{print $0}'
$
```

If the *e* character doesn’t appear in the text, or as long as it appears only once in the text, the pattern matches.

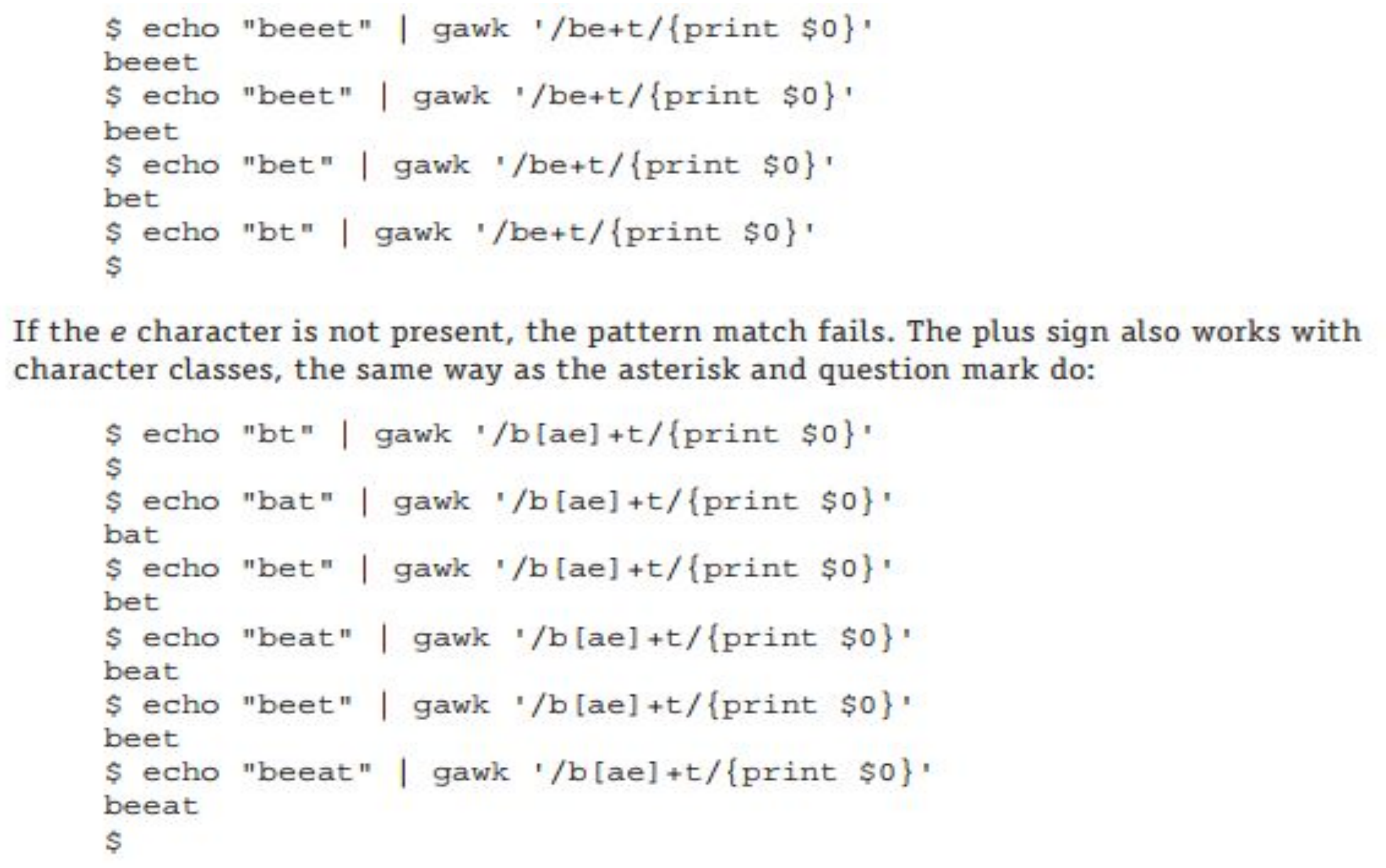
As with the asterisk, you can use the question mark symbol along with a character class:

```
$ echo "bt" | gawk '/b[ae]?t/{print $0}'
bt
$ echo "bat" | gawk '/b[ae]?t/{print $0}'
bat
$ echo "bot" | gawk '/b[ae]?t/{print $0}'
$
$ echo "bet" | gawk '/b[ae]?t/{print $0}'
bet
$ echo "baet" | gawk '/b[ae]?t/{print $0}'
$
$ echo "beat" | gawk '/b[ae]?t/{print $0}'
$
$ echo "beet" | gawk '/b[ae]?t/{print $0}'
$
```

If zero or one character from the character class appears, the pattern match passes. However, if both characters appear, or if one of the characters appears twice, the pattern match fails.

- **The plus sign**

  The plus sign is another pattern symbol that’s similar to the asterisk, but with a different twist than the question mark. The plus sign indicates that the **preceding character can appear one or more times**, but must be present at least once. The pattern doesn’t match if the character is not present:



```
$ echo "beeet" | gawk '/be+t/{print $0}'
beeet
$ echo "beet" | gawk '/be+t/{print $0}'
beet
$ echo "bet" | gawk '/be+t/{print $0}'
bet
$ echo "bt" | gawk '/be+t/{print $0}'
$
```

If the *e* character is not present, the pattern match fails. The plus sign also works with character classes, the same way as the asterisk and question mark do:

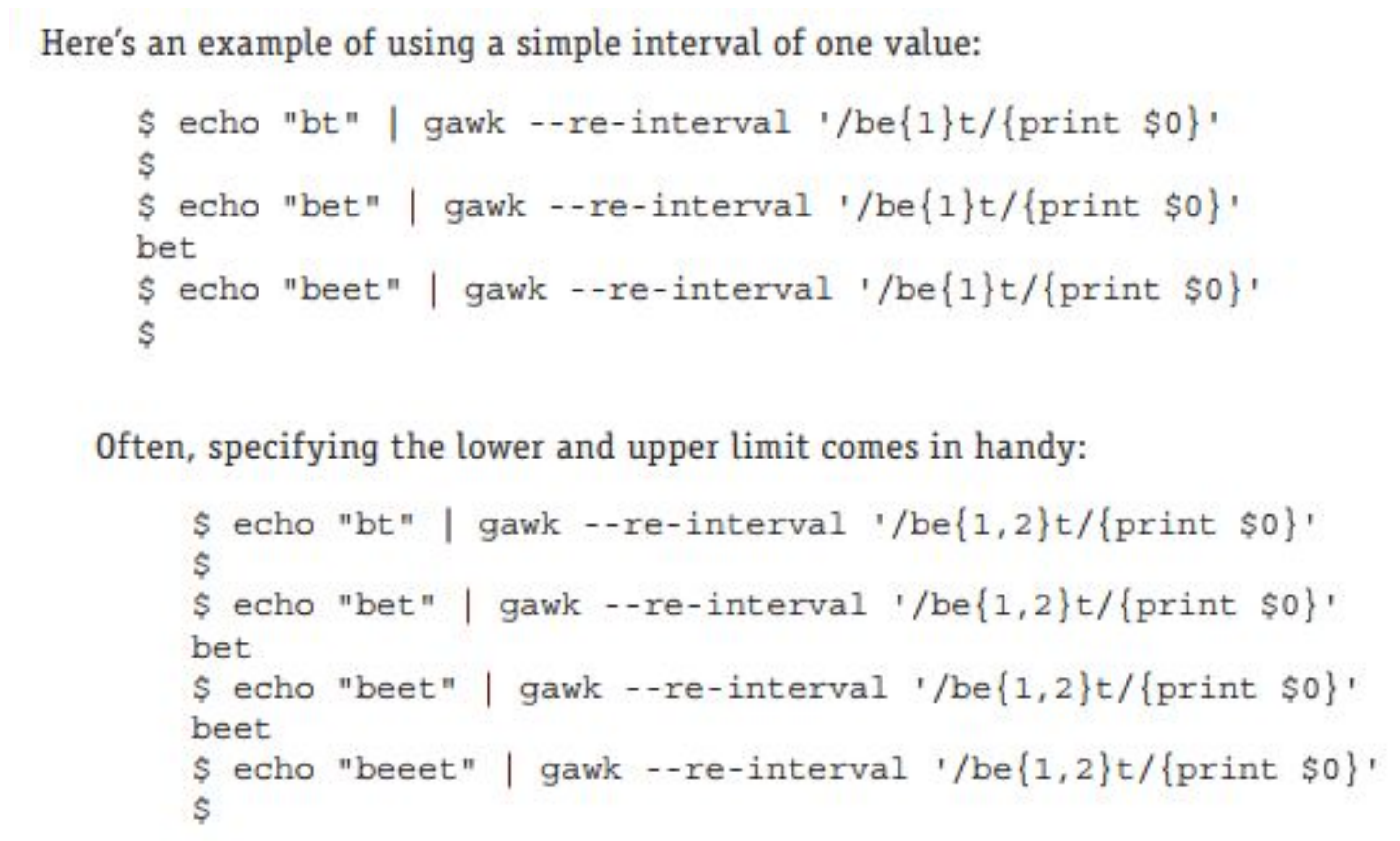
```
$ echo "bt" | gawk '/b[ae]+t/{print $0}'
$
$ echo "bat" | gawk '/b[ae]+t/{print $0}'
bat
$ echo "bet" | gawk '/b[ae]+t/{print $0}'
bet
$ echo "beat" | gawk '/b[ae]+t/{print $0}'
beat
$ echo "beet" | gawk '/b[ae]+t/{print $0}'
beet
$ echo "beeat" | gawk '/b[ae]+t/{print $0}'
beeat
$
```

- **Using braces**

  Curly braces are available in ERE to allow you to specify a limit on a repeatable regular expression. This is often referred to as an *interval.* You can express the interval in two formats:

  ■ `m`: The regular expression appears **exactly *m* times.**

  ■ `m,n`: The regular expression appears **at least *m* times, but no more than *n* times.**



Here's an example of using a simple interval of one value:

```
$ echo "bt" | gawk --re-interval '/be{1}t/{print $0}'
$
$ echo "bet" | gawk --re-interval '/be{1}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1}t/{print $0}'
$
```

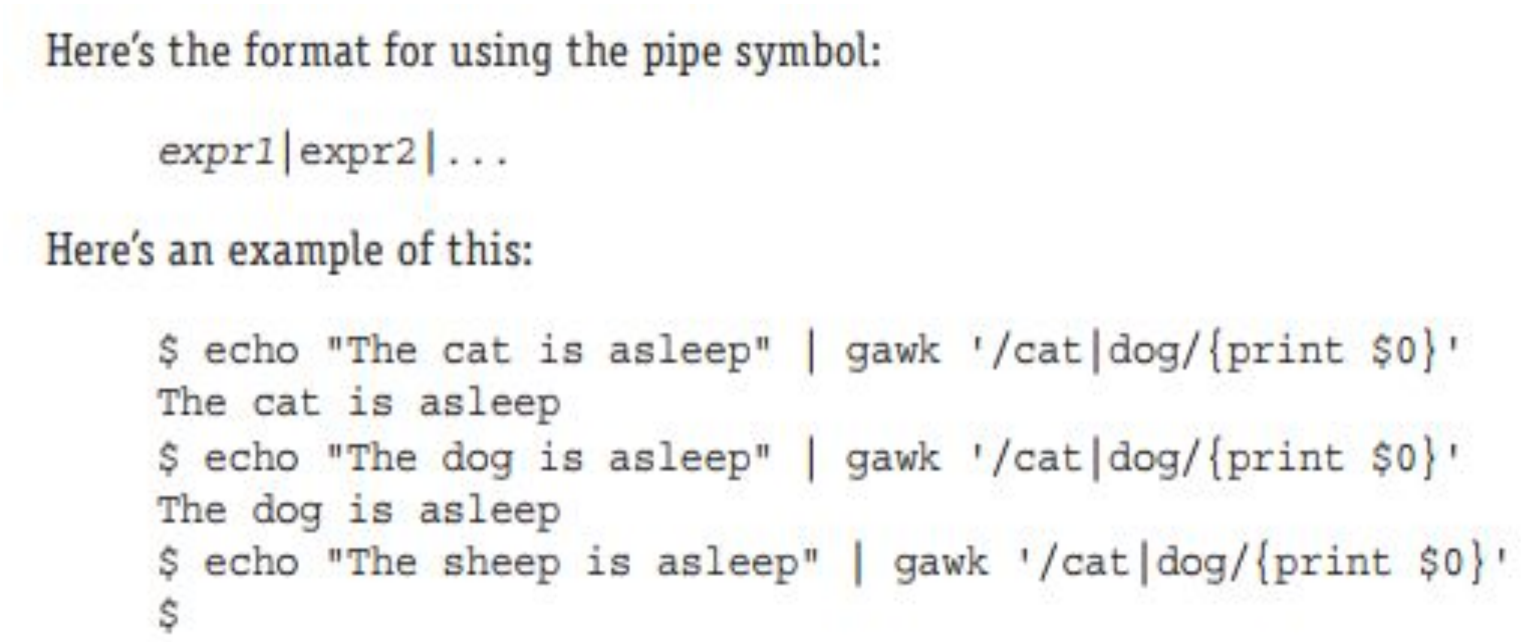
Often, specifying the lower and upper limit comes in handy:

```
$ echo "bt" | gawk --re-interval '/be{1,2}t/{print $0}'
$
$ echo "bet" | gawk --re-interval '/be{1,2}t/{print $0}'
bet
$ echo "beet" | gawk --re-interval '/be{1,2}t/{print $0}'
beet
$ echo "beeet" | gawk --re-interval '/be{1,2}t/{print $0}'
$
```

The interval pattern match also applies to character classes:

```
$ echo "bt" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "bat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bat
$ echo "bet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
bet
$ echo "beat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beat
$ echo "beet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
beet
$ echo "beeat" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
$ echo "baeet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
```

- **The pipe symbol:**
- The pipe symbol allows to you to **specify two or more patterns that the regular expression engine uses in a logical `OR` formula** when examining the data stream. If any of the patterns match the data stream text, the text passes. If none of the patterns match, the data stream text fails.



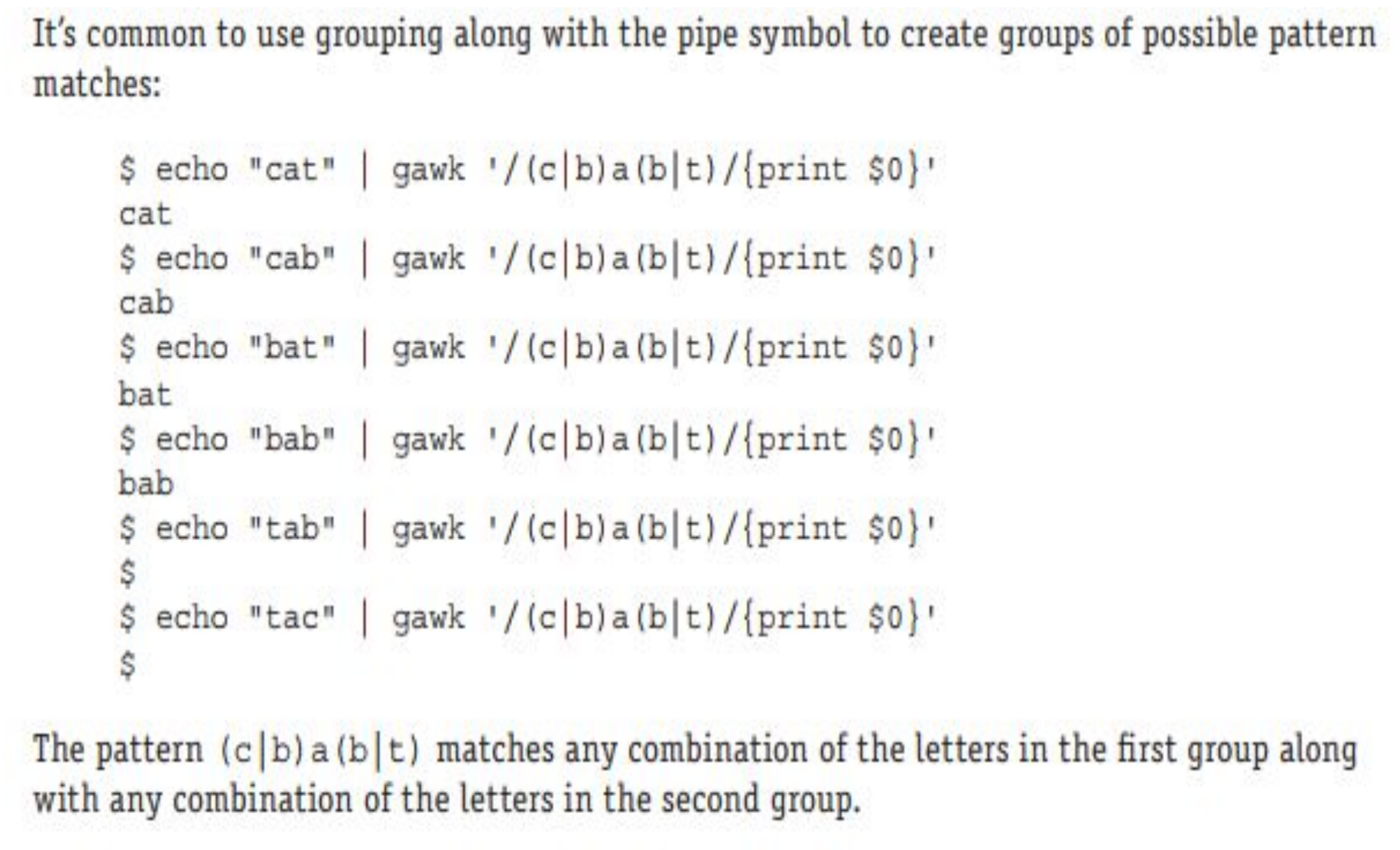
Here's the format for using the pipe symbol:

```
expr1|expr2|...
```

Here's an example of this:

```
$ echo "The cat is asleep" | gawk '/cat|dog/{print $0}'
The cat is asleep
$ echo "The dog is asleep" | gawk '/cat|dog/{print $0}'
The dog is asleep
$ echo "The sheep is asleep" | gawk '/cat|dog/{print $0}'
$
```

- **Grouping expressions:**

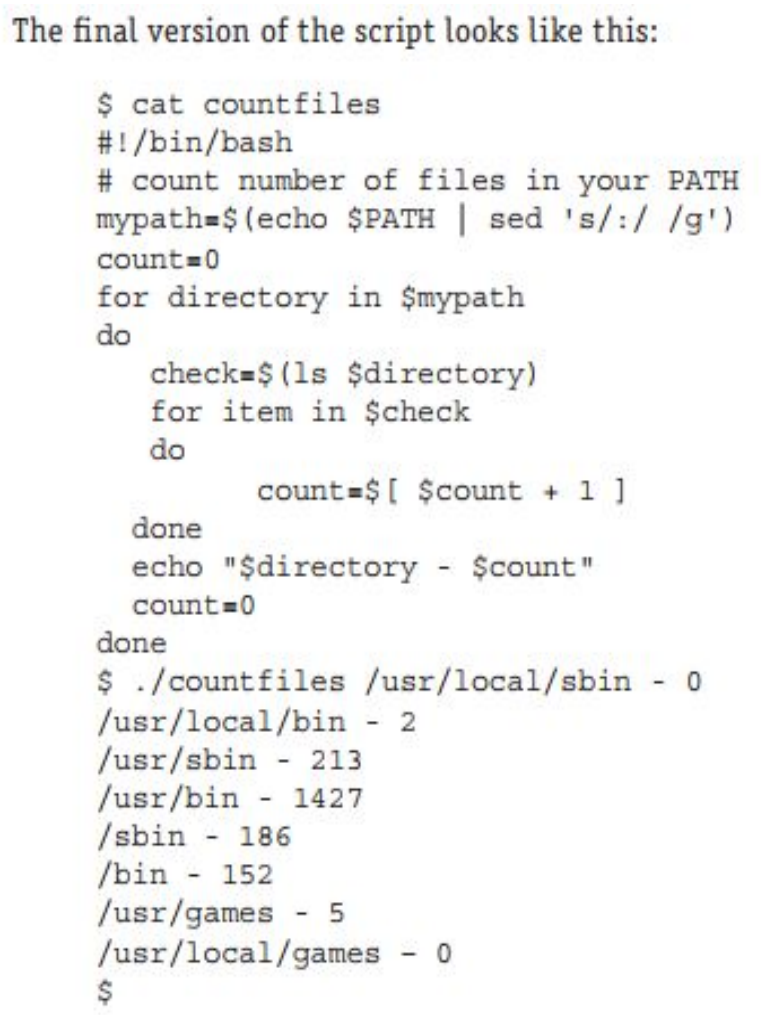


It's common to use grouping along with the pipe symbol to create groups of possible pattern matches:

```
$ echo "cat" | gawk '/(c|b)a(b|t)/{print $0}'
cat
$ echo "cab" | gawk '/(c|b)a(b|t)/{print $0}'
cab
$ echo "bat" | gawk '/(c|b)a(b|t)/{print $0}'
bat
$ echo "bab" | gawk '/(c|b)a(b|t)/{print $0}'
bab
$ echo "tab" | gawk '/(c|b)a(b|t)/{print $0}'
$
$ echo "tac" | gawk '/(c|b)a(b|t)/{print $0}'
$
```

The pattern `(c|b)a(b|t)` matches any combination of the letters in the first group along with any combination of the letters in the second group.

- **Counting directory files**
  To start things out, let’s look at a shell script that counts the executable files that are present in the directories defined in your `PATH` environment variable. To do that, you need to parse out the `PATH` variable into separate directory names.



The final version of the script looks like this:

```
$ cat countfiles
#!/bin/bash
# count number of files in your PATH
mypath=$(echo $PATH | sed 's/:/ /g')
count=0
for directory in $mypath
do
   check=$(ls $directory)
   for item in $check
   do
         count=$[ $count + 1 ]
  done
  echo "$directory - $count"
  count=0
done
$ ./countfiles /usr/local/sbin - 0
/usr/local/bin - 2
/usr/sbin - 213
/usr/bin - 1427
/sbin - 186
/bin - 152
/usr/games - 5
/usr/local/games - 0
$
```

- **Validating a phone number:**

  A common data validation application checks phone numbers. Often, data entry forms request phone numbers, and often customers fail to enter a properly formatted phone number. People in the United States use several common ways to display a phone number:

  ```
  (123)456-7890
  (123) 456-7890
  123-456-7890

  123.456.7890
  ```

  In this example, there may or may not be a left parenthesis in the phone number. This can be matched by using the pattern:

  ```
  ^\(?
  ```

- The **caret is used to indicate the beginning of the data**. Because the left parenthesis is a special character, you must escape it to use it as a normal character. **The question mark indicates that the left parenthesis may or may not appear in the data to match.**

- `[2-9][0-9]{2}`

  This requires that the first character be a digit between 2 and 9, followed by any two digits. After the area code, the ending right parenthesis may or may not appear:

  ```
  \)?
  ```

- After the area code, there can be a space, no space, a dash, or a dot. You can group those using a character group along with the pipe symbol:

  ```
  (| |-|\.)
  ```

  The very first pipe symbol appears immediately after the left parenthesis to match the no space condition. You must use the escape character for the dot; otherwise, it is interpreted to match any character.

- Next is the three-digit phone exchange number. Nothing special is required here:

  ```
  [0-9]{3}
  ```

  After the phone exchange number, you must match a space, a dash, or a dot (this time you don't have to worry about matching no space because there must be at least a space between the phone exchange number and the rest of the number):

  ```
  ( |-|\.)
  ```

  Then to finish things off, you must match the four-digit local phone extension at the end of the string:

  ```
  [0-9]{4}$
  ```

  Putting the entire pattern together results in this:

  ```
  ^\(?[2-9][0-9]{2}\)?(| |-|\.)[0-9]{3}( |-|\.)[0-9]{4}$
  ```

Advanced sed

- **Looking at Multiline Commands**
  As the `sed` editor reads a data stream, it divides the data into lines based on the presence of newline characters. The `sed` editor handles each data line one at a time, processing the defined script commands on the data line, and then moving on to the next line and repeating the processing.

- The `sed` editor includes three special commands that you can use to process multiline text:

  ■ `N` adds the next line in the data stream to create a multiline group for processing.

  ■ `D` deletes a single line in a multiline group.

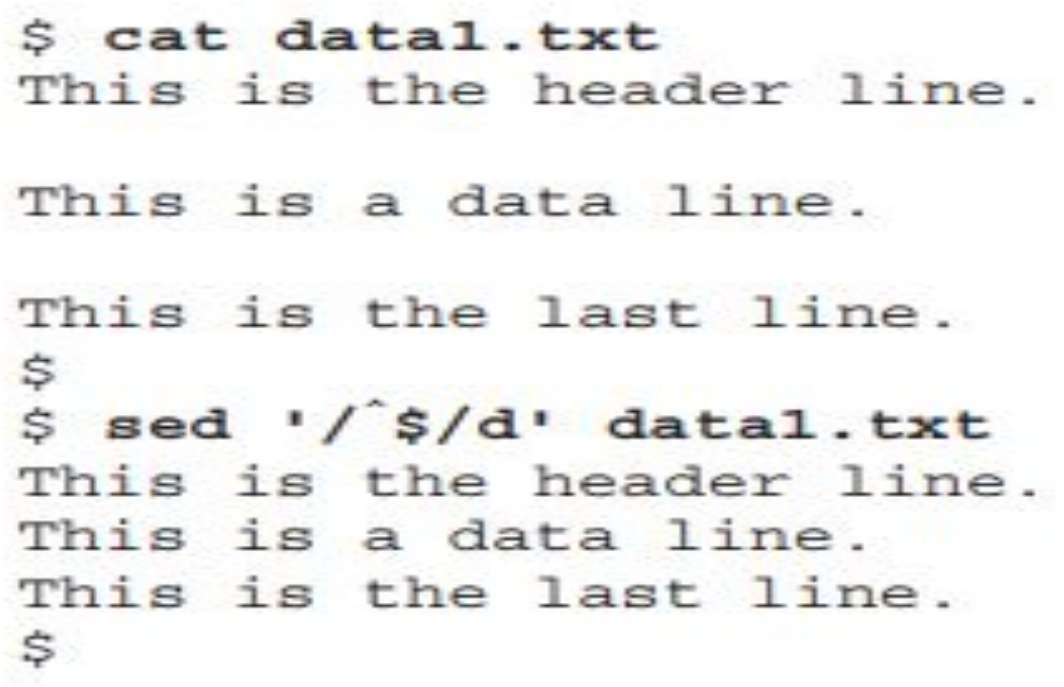
  ■ `P` prints a single line in a multiline group.

  The following sections examine these multiline commands more closely and demonstrate how you can use them in your scripts.

- **Navigating the next command:**

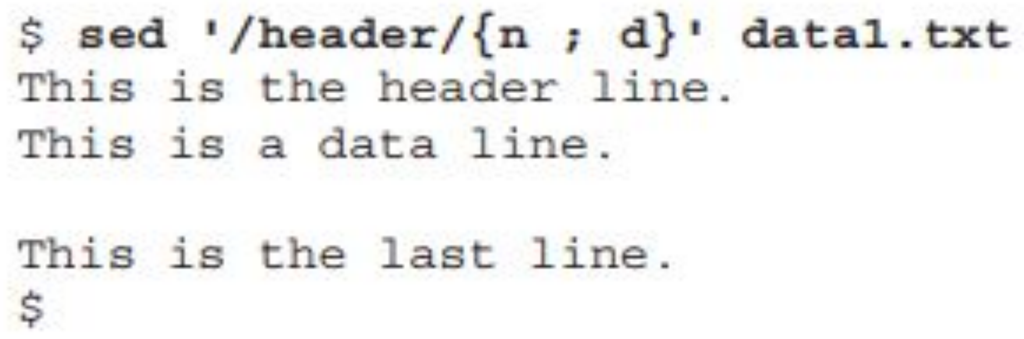
Using the single-line next command:

The lowercase `n` command tells the `sed` editor to move to the next line of text in the data stream, without going back to the beginning of the commands. Remember that normally the `sed` editor processes all the defined commands on a line before moving to the next line of text in the data stream. The single-line `next` command alters this flow.
In this example, you have a data file that contains five lines, two of which are empty. The goal is to remove the blank line after the header line but leave the blank line before the last line intact. If you write a `sed` script to just remove blank lines, you remove both blank lines:



```
$ cat data1.txt
This is the header line.

This is a data line.

This is the last line.
$
$ sed '/^$/d' data1.txt
This is the header line.
This is a data line.
This is the last line.
$
```

- In this next example, the script looks for a unique line that contains the word `header`. After the script identifies that line, the `n` command moves the `sed` editor to the next line of text, which is the empty line.

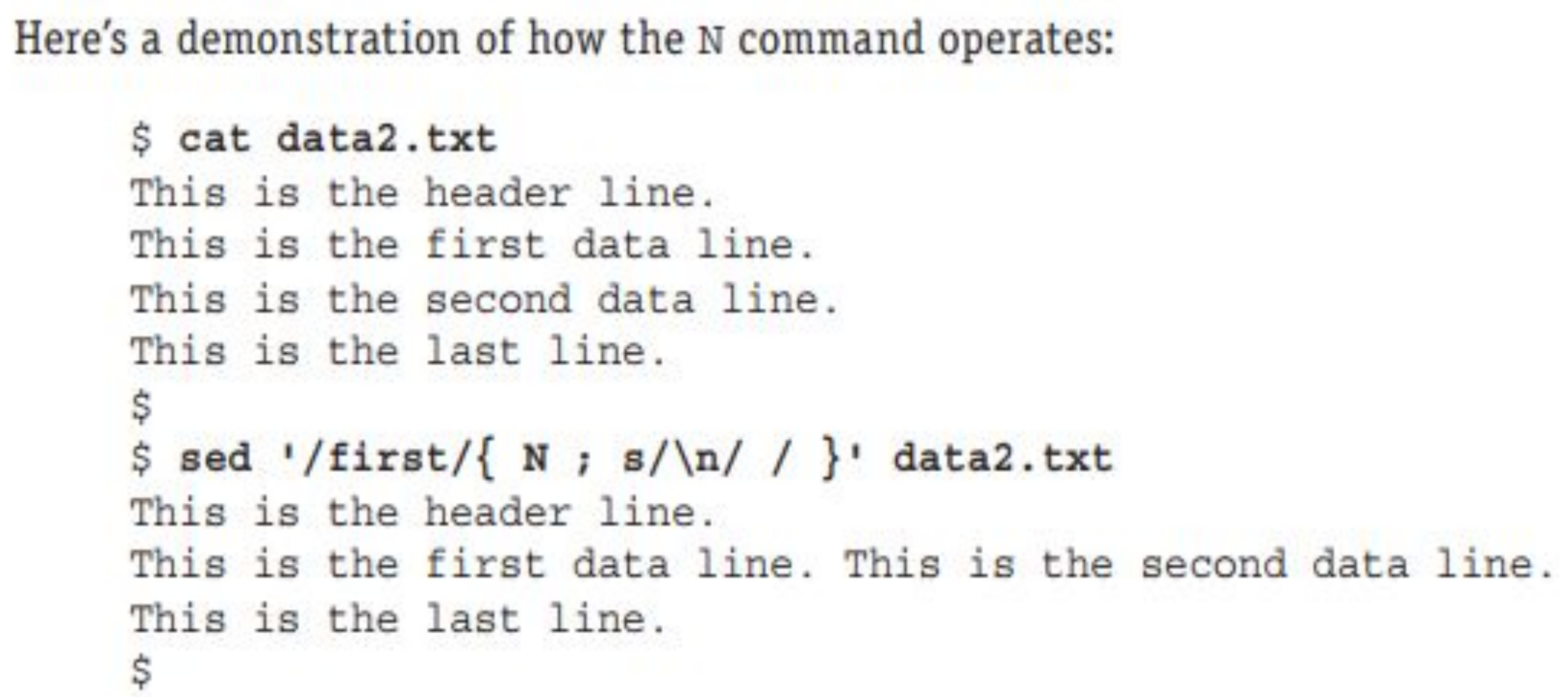


```
$ sed '/header/{n ; d}' data1.txt
This is the header line.
This is a data line.

This is the last line.
$
```

- At that point, the `sed` editor continues processing the command list, which uses the `d` command to delete the empty line. When the `sed` editor reaches the end of the command script, it reads the next line of text from the data stream and starts processing commands from the top of the command script. The `sed` editor does not find another line with the word `header`; thus, no further lines are deleted.

- **Combining lines of text**:
- Now that you’ve seen the single-line `next` command, you can look at the multiline version. The single-line `next` command moves the next line of text from the data stream into the processing space (called the *`pattern space`*) of the `sed` editor. The multiline version of the `next` command (which uses a capital `N`) adds the next line of text to the text already in the pattern space.

- The `sed` editor script searches for the line of text that contains the word “first” in it. When it finds the line, it uses the `N` command to combine the next line with that line. It then uses the `substitution` command (`s`) to replace the newline character with a space. The result is that the two lines in the text file appear as one line in the `sed` editor output.



Here’s a demonstration of how the N command operates:

```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed '/first/{ N ; s/\n/ / }' data2.txt
This is the header line.
This is the first data line. This is the second data line.
This is the last line.
$
```

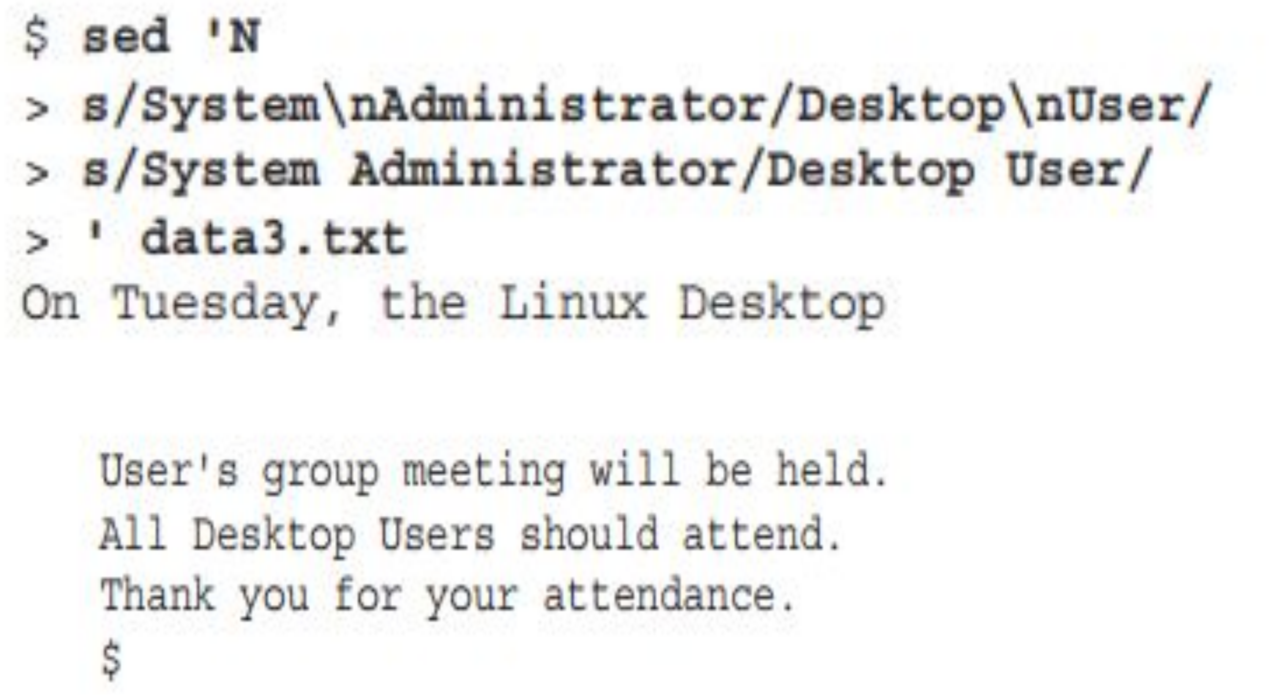
```
$ cat data3.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All System Administrators should attend.
Thank you for your attendance.
$
$ sed 'N ; s/System Administrator/Desktop User/' data3.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All Desktop Users should attend.
Thank you for your attendance.
$
```

The `substitution` command is looking for the specific two-word phrase `System Administrator` in the text file. In the single line where the phrase appears, everything is fine; the `substitution` command can replace the text. But in the situation where the phrase is split between two lines, the `substitution` command doesn't recognize the matching pattern.

The `N` command helps solve this problem:

```
$ sed 'N ; s/System.Administrator/Desktop User/' data3.txt
On Tuesday, the Linux Desktop User's group meeting will be held.
All Desktop Users should attend.
Thank you for your attendance.
$
```

- To solve this problem, you can use two substitution commands in the sed editor script, one to match the multiline occurrence and one to match the single-line occurrence:

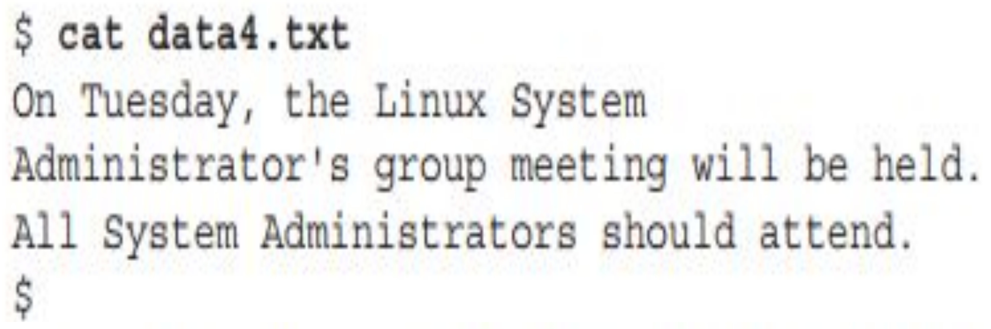


```
$ sed 'N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data3.txt
On Tuesday, the Linux Desktop

  User's group meeting will be held.
  All Desktop Users should attend.
  Thank you for your attendance.
  $
```

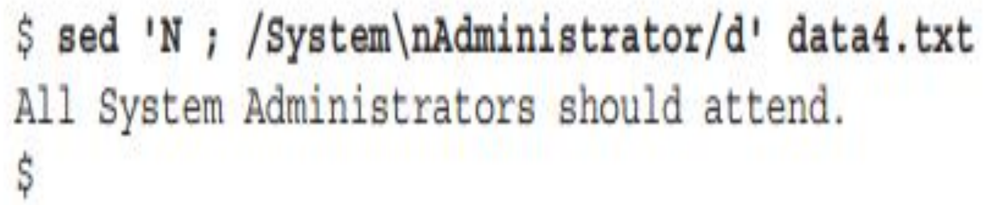
- The first substitution command specifically looks for the newline character between the two search words and includes it in the replacement string. This allows you to add the newline character in the same place in the new text.

```
$ cat data4.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All System Administrators should attend.
$
$ sed 'N
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All System Administrators should attend.
$
```

- **Navigating the multiline delete command:**
  The `sed` editor uses it to delete the current line in the pattern space. If you’re working with the `N` command, however, you must be careful when using the single-line `delete` command:



```
$ cat data4.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All System Administrators should attend.
$
```



```
$ sed 'N ; /System\nAdministrator/d' data4.txt
All System Administrators should attend.
$
```

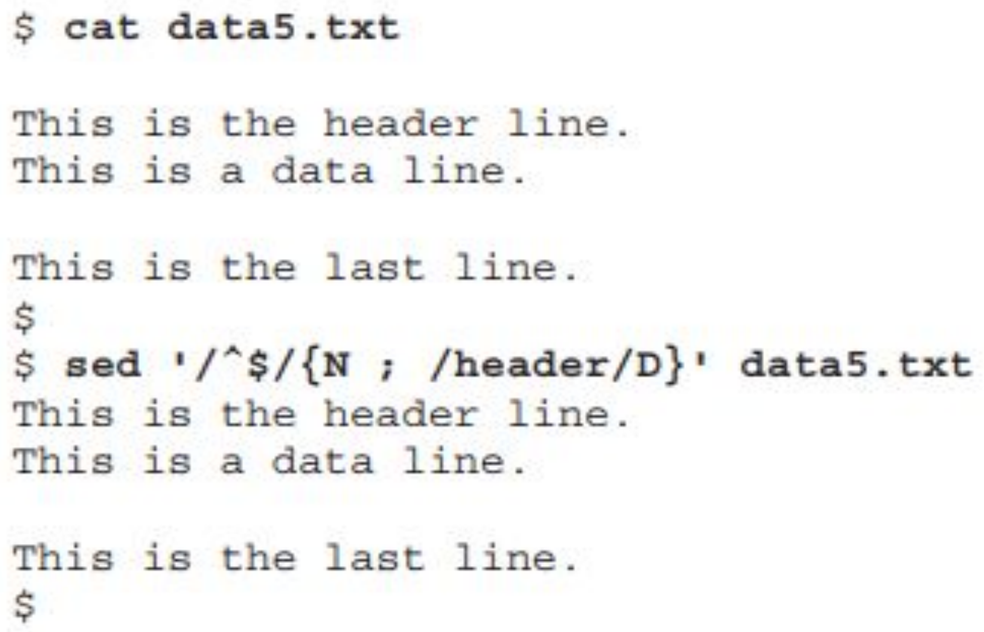
- The `delete` command looked for the words `System` and `Administrator` in separate lines and deleted both of the lines in the pattern space. This may or may not have been what you intended.

- The `sed` editor provides the multiline `delete` command (`D`), which deletes only the first line in the pattern space. It removes all characters up to and including the newline character:

```
$ sed 'N ; /System\nAdministrator/D' data4.txt
Administrator's group meeting will be held.
All System Administrators should attend.
$
```

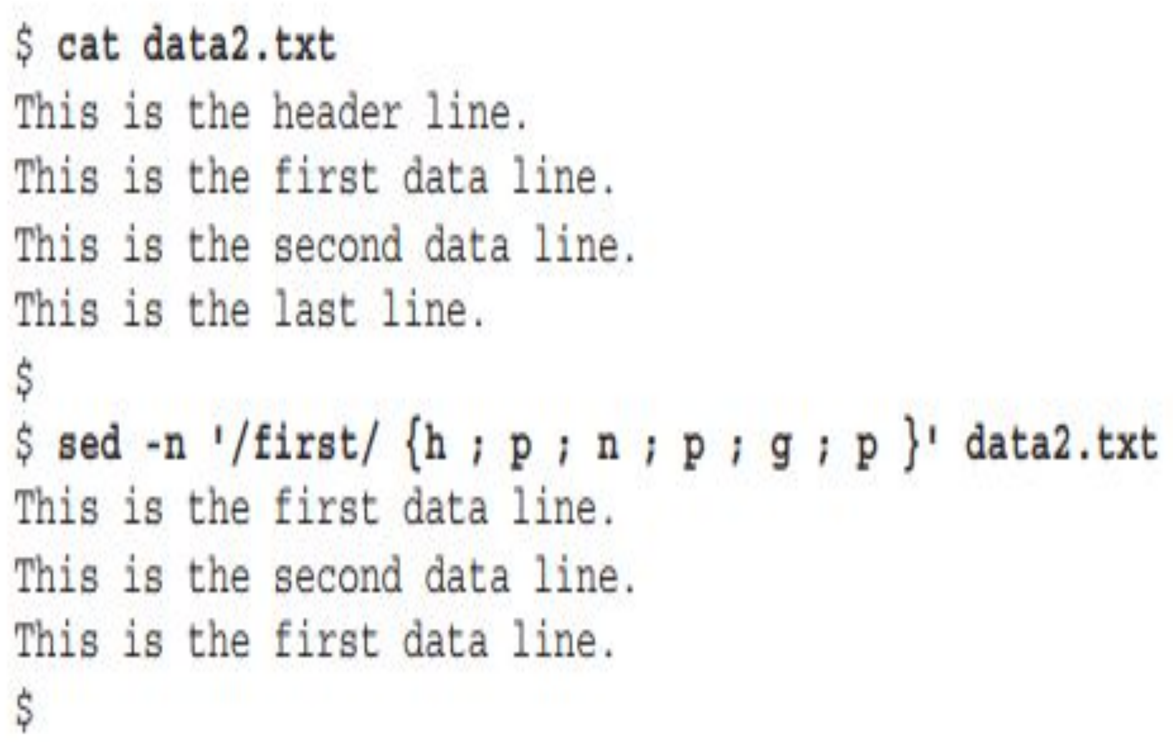
  The second line of text, added to the pattern space by the `N` command, remains intact. This comes in handy if you need to remove a line of text that appears before a line that you find a data string in.

- Here's an example of removing a blank line that appears before the first line in a data stream:



```
$ cat data5.txt

This is the header line.
This is a data line.

This is the last line.
$
$ sed '/^$/{N ; /header/D}' data5.txt
This is the header line.
This is a data line.

This is the last line.
$
```

- **Holding Space:**
  The *pattern space* is an active buffer area that holds the text examined by the `sed` editor while it processes commands. However, it isn’t the only space available in the `sed` editor for storing text. The `sed` editor utilizes another buffer area called the *`hold space`*. You can use the hold space to temporarily hold lines of text while working on other lines in the pattern space

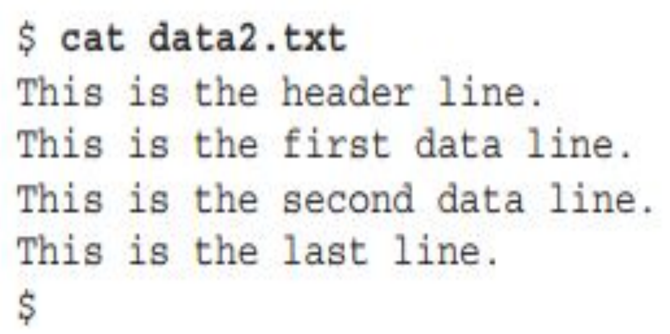


```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed -n '/first/ {h ; p ; n ; p ; g ; p }' data2.txt
This is the first data line.
This is the second data line.
This is the first data line.
$
```

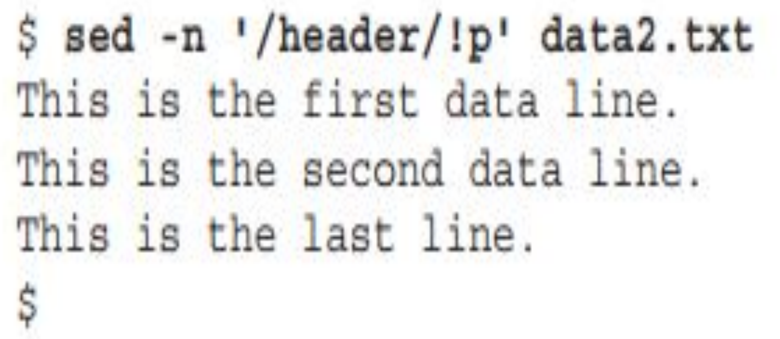
- **Look at the preceding code example step by step:**

  1. The `sed` script uses a regular expression in the address to filter the line containing the word `first`.
  2. When the line containing the word `first` appears, the initial command in `{}`, the `h` command, places the line in the hold space.
  3. The next command, the `p` command, prints the contents of the pattern space, which is still the first data line.
  4. The `n` command retrieves the next line in the data stream(`This is the second data line`) and places it in the pattern space.
  5. The `p` command prints the contents of the pattern space, which is now the second data line.
  6. The `g` command places the contents of the hold space (`This is the first data line`) back into the pattern space, replacing the current text.
  7. The `p` command prints the current contents of the pattern space, which is now back to the first data line.

- **Negating a Command:**
  The exclamation mark command (`!`) is used to negate a command. This means in situations where the command would normally have been activated, it isn't. Here's an example demonstrating this feature:



```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
```



```
$ sed -n '/header/!p' data2.txt
This is the first data line.
This is the second data line.
This is the last line.
$
```

- The normal `p` command would have printed only the line in the `data2` file that contained the word `header`. By adding the exclamation mark, the opposite happens — all lines in the file are printed except the one that contained the word header.

```
$ cat data4.txt
On Tuesday, the Linux System
Administrator's group meeting will be held.
All System Administrators should attend.
$
```

```
$ sed 'N;
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
```

```
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All System Administrators should attend.
$
$ sed '$!N;
> s/System\nAdministrator/Desktop\nUser/
> s/System Administrator/Desktop User/
> ' data4.txt
On Tuesday, the Linux Desktop
User's group meeting will be held.
All Desktop Users should attend.
$
```

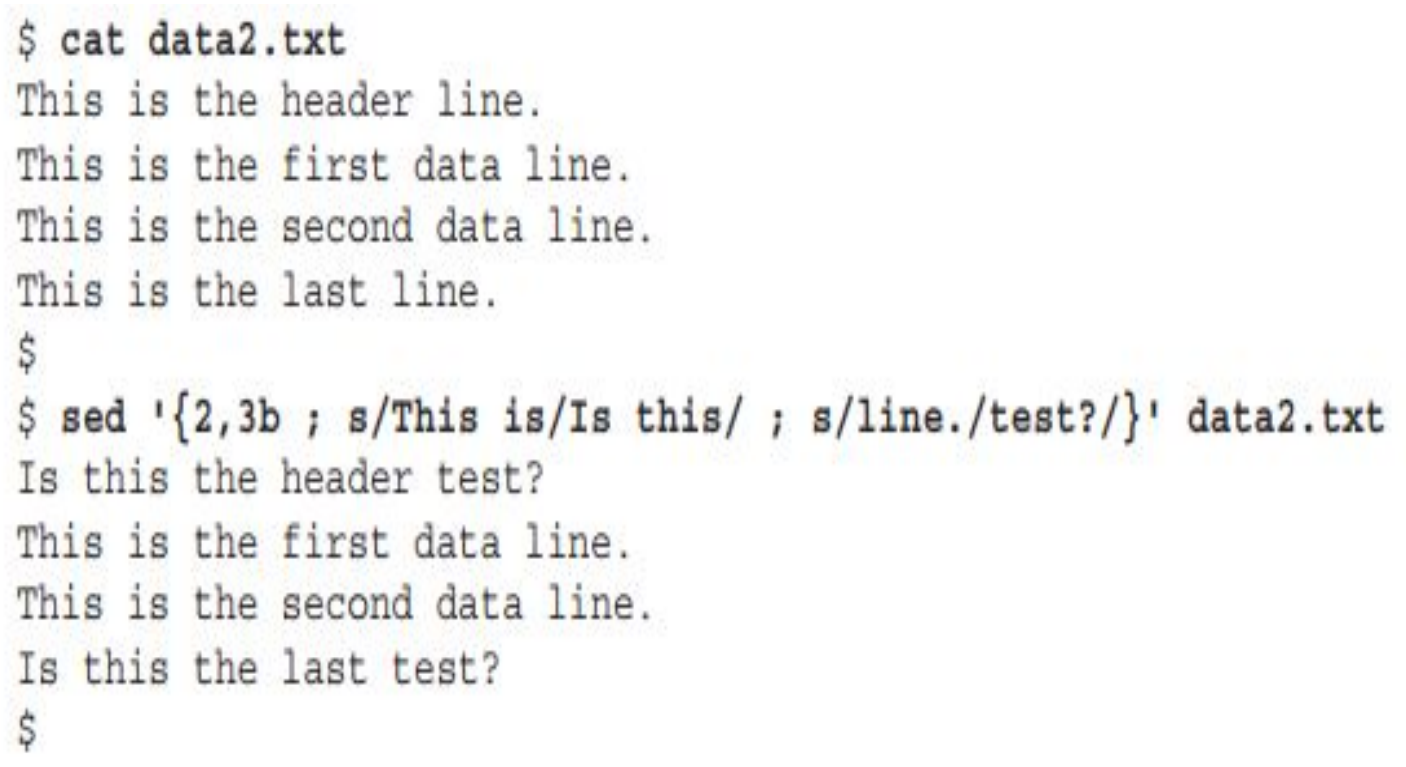
- **Changing the Flow:**

Branching:

Here’s the format of the `branch` command:

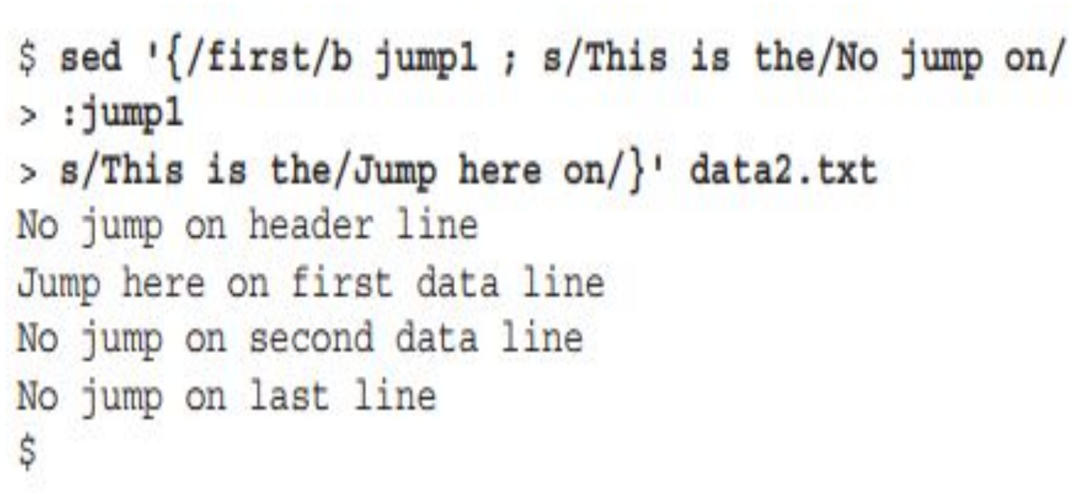
`[`*`address`*`]b [`*`label`*`]`

The `address` parameter determines which line or lines of data trigger the `branch` command. The `label` parameter defines the location to branch to. If the `label` parameter is not present, the `branch` command proceeds to the end of the script.



```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
$ sed '{2,3b ; s/This is/Is this/ ; s/line./test?/}' data2.txt
Is this the header test?
This is the first data line.
This is the second data line.
Is this the last test?
$
```

- To specify the label, just add it after the `b` command. Using labels allows you to skip commands that match the `branch` address but still process other commands in the script:



```
$ sed '{/first/b jump1 ; s/This is the/No jump on/
> :jump1
> s/This is the/Jump here on/}' data2.txt
No jump on header line
Jump here on first data line
No jump on second data line
No jump on last line
$
```

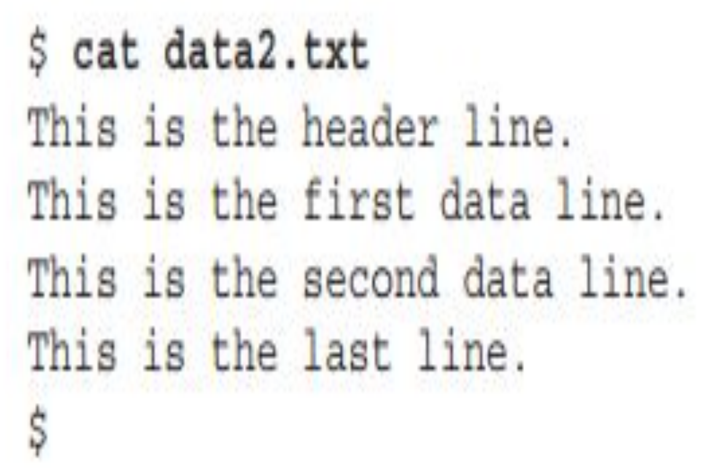
- The `branch` command specifies that the program should jump to the script line labeled `jump1` if the matching text “first” appears in the line. If the `branch` command pattern doesn’t match, the `sed` editor continues processing commands in the script, including the command after the `branch` label. (Thus, all three substitution commands are processed on lines that don’t match the branch pattern.)

• Testing:

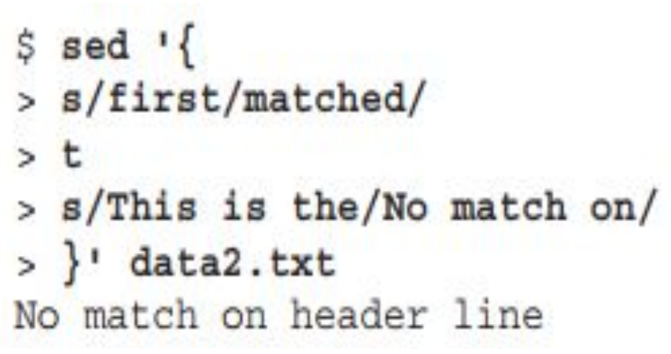
- If the `substitution` command successfully matches and substitutes a pattern, the `test` command branches to the specified label. If the `substitution` command doesn't match the specified pattern, the `test` command doesn't branch. The `test` command uses the same format as the `branch` command:

- `[`*`address`*`]t [`*`label`*`]`
  Like the `branch` command, if you don't specify a label, `sed` branches to the end of the script if the test succeeds.

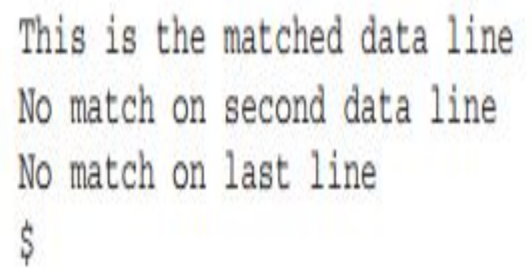
- The `test` command provides a cheap way to perform a basic `if-then` statement on the text in the data stream. For example, if you don't need to make a substitution if another substitution was made, the `test` command can help:



```
$ cat data2.txt
This is the header line.
This is the first data line.
This is the second data line.
This is the last line.
$
```



```
$ sed '{
> s/first/matched/
> t
> s/This is the/No match on/
> }' data2.txt
No match on header line
```



```
This is the matched data line
No match on second data line
No match on last line
$
```

The first `substitution` command looks for the pattern text `first`. If it matches the pattern in the line, it replaces the text, and the `test` command jumps over the second `substitution` command. If the first `substitution` command doesn't match the pattern, the second `substitution` command is processed.

- **Replacing via a Pattern:**
- For example, say that you want to place double quotation marks around a word you match in a line. That’s simple enough if you’re just looking for one word in the pattern to match:

-
  ```
  $ echo "The cat sleeps in his hat." | sed 's/cat/"cat"/'
  The "cat" sleeps in his hat.
  $
  ```

  But what if you use a wildcard character (.) in the pattern to match more than one word?

  ```
  $ echo "The cat sleeps in his hat." | sed 's/.at/".at"/g'
  The ".at" sleeps in his ".at".
  $
  ```

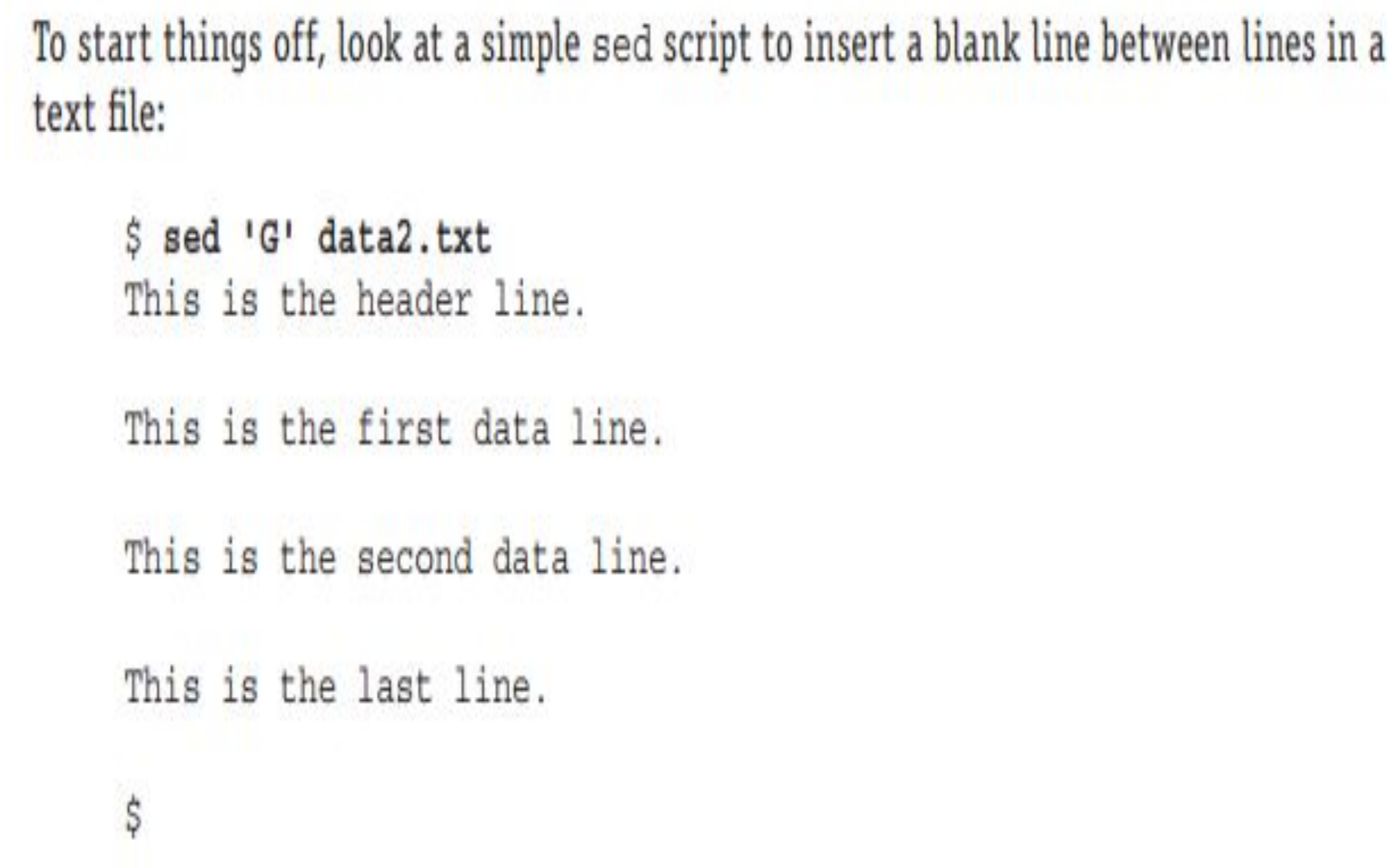
  The `sed` editor has a solution for you. The ampersand symbol (`&`) is used to represent the matching pattern in the `substitution` command. Whatever text matches the pattern defined, you can use the ampersand symbol to recall it in the replacement pattern. This lets you manipulate whatever word matches the pattern defined:

-
  ```
  $ echo "The cat sleeps in his hat." | sed 's/.at/"&"/g'
  The "cat" sleeps in his "hat".
  $
  ```

  When the pattern matches the word cat, “cat” appears in the substituted word. When it matches the word hat, “hat” appears in the substituted word.

• **Creating sed Utilities:**

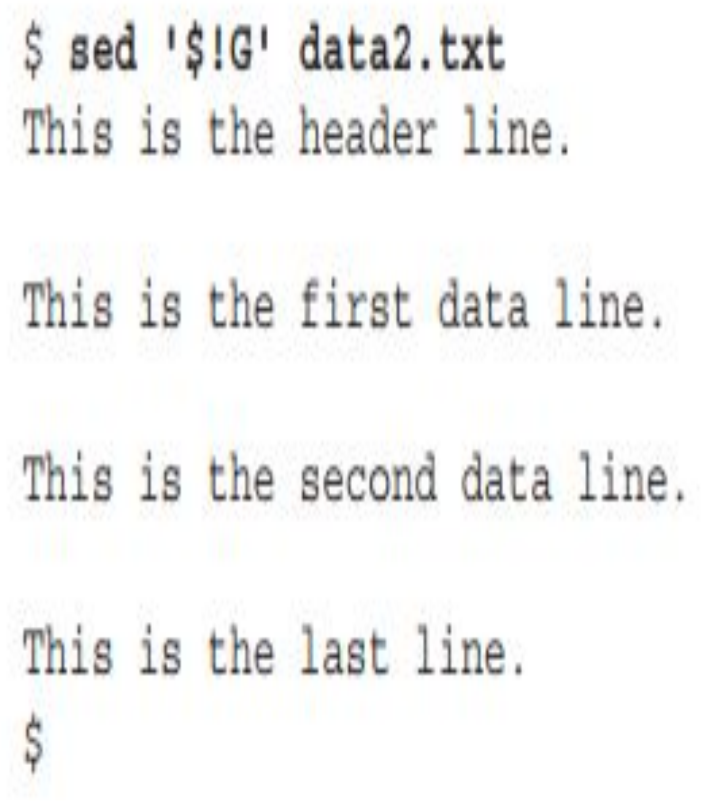
Spacing with double lines The key to this trick is the default value of the hold space. Remember that the `G` command simply appends the contents of the hold space to the current pattern space contents. When you start the `sed` editor, the hold space contains an empty line. By appending that to an existing line, you create a blank line after the existing line.



To start things off, look at a simple sed script to insert a blank line between lines in a text file:

```
$ sed 'G' data2.txt
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.

$
```

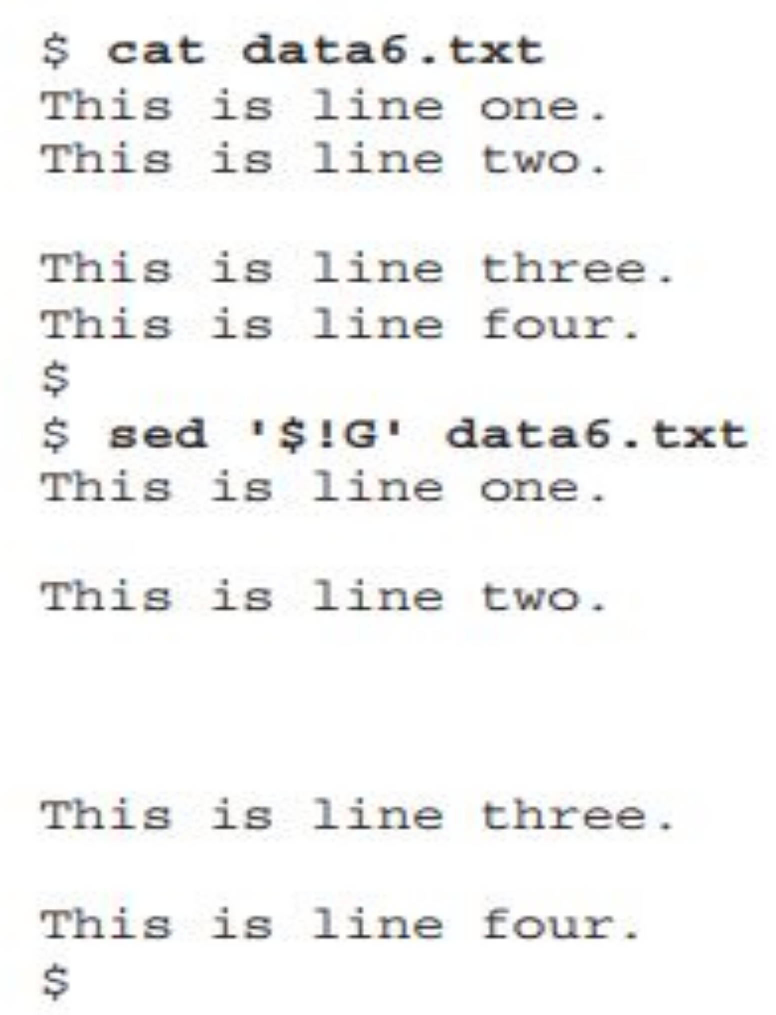
- You may have noticed that this script also adds a blank line to the last line in the data stream, producing a blank line at the end of the file. If you want to get rid of this, you can use the negate symbol and the last line symbol to ensure that the script doesn't add the blank line to the last line of the data stream:



```
$ sed '$!G' data2.txt
This is the header line.

This is the first data line.

This is the second data line.

This is the last line.
$
```

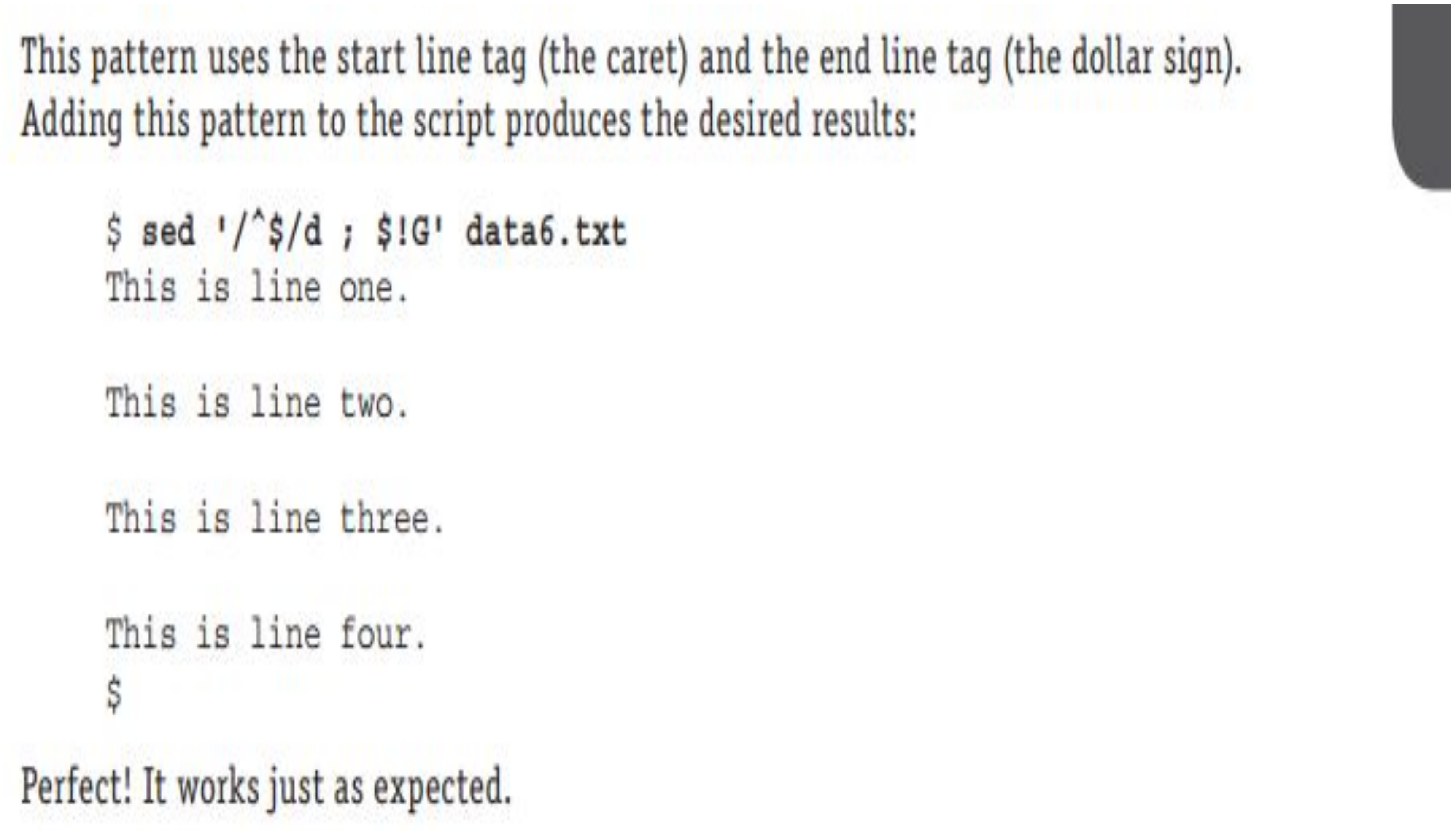
- **Spacing files that may have blanks:**

To take double spacing one step further, what if the text file already has a few blank lines, but you want to double space all the lines? If you use the previous script, you'll get some areas that have too many blank lines, because each existing blank line gets doubled:



```
$ cat data6.txt
This is line one.
This is line two.

This is line three.
This is line four.
$
$ sed '$!G' data6.txt
This is line one.

This is line two.



This is line three.

This is line four.
$
```

This pattern uses the start line tag (the caret) and the end line tag (the dollar sign). Adding this pattern to the script produces the desired results:



This pattern uses the start line tag (the caret) and the end line tag (the dollar sign). Adding this pattern to the script produces the desired results:

```
$ sed '/^$/d ; $!G' data6.txt
This is line one.

This is line two.

This is line three.

This is line four.
$
```

Perfect! It works just as expected.

- **Deleting lines:**

- Another useful utility for the `sed` editor is to remove unwanted blank lines in a data stream. It’s easy to remove all the blank lines from a data stream, but it takes a little ingenuity to selectively remove blank lines.

- The easiest way to remove consecutive blank lines is to check the data stream using a range address. Unit III, showed you how to use ranges in addresses, including how to incorporate patterns in the address range. The `sed` editor executes the command for all lines that match within the specified address range.

- The key to removing consecutive blank lines is to create an address range that includes a non-blank line and a blank line. If the `sed` editor comes across this range, it shouldn’t delete the line. However, for lines that don’t match that range (two or more blank lines in a row), it should delete the lines.

  Here’s the script to do this:

```
/./,/^$/!d
```

The range is /./ to `/^$/`. The start address in the range matches any line that contains at least one character. The end address in the range matches a blank line. Lines within this range aren’t deleted.

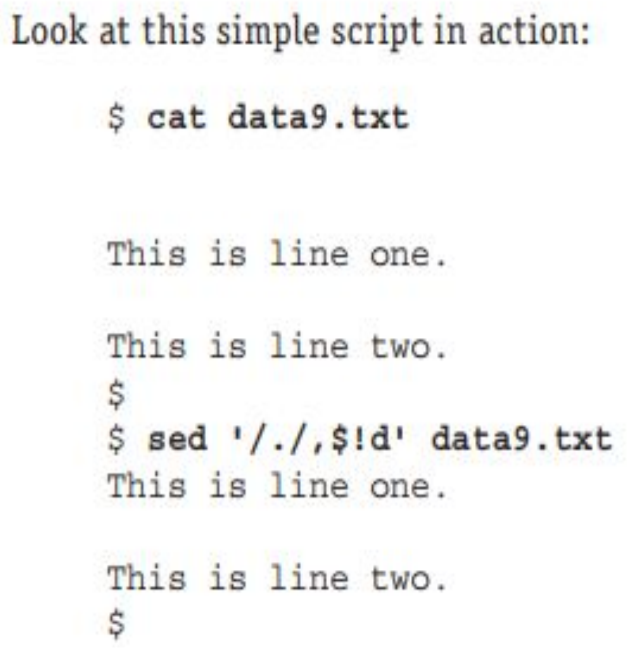
Here's the script in action:

```
$ cat data8.txt
This is line one.


This is line two.

This is line three.



This is line four.

$
$ sed '/./,/^$/!d' data8.txt
This is line one.

This is line two.

This is line three.

This is line four.
$
```

- Removing blank lines from the top of a data stream is not a difficult task. Here’s the script that accomplishes that function:

  `/./,$!d`

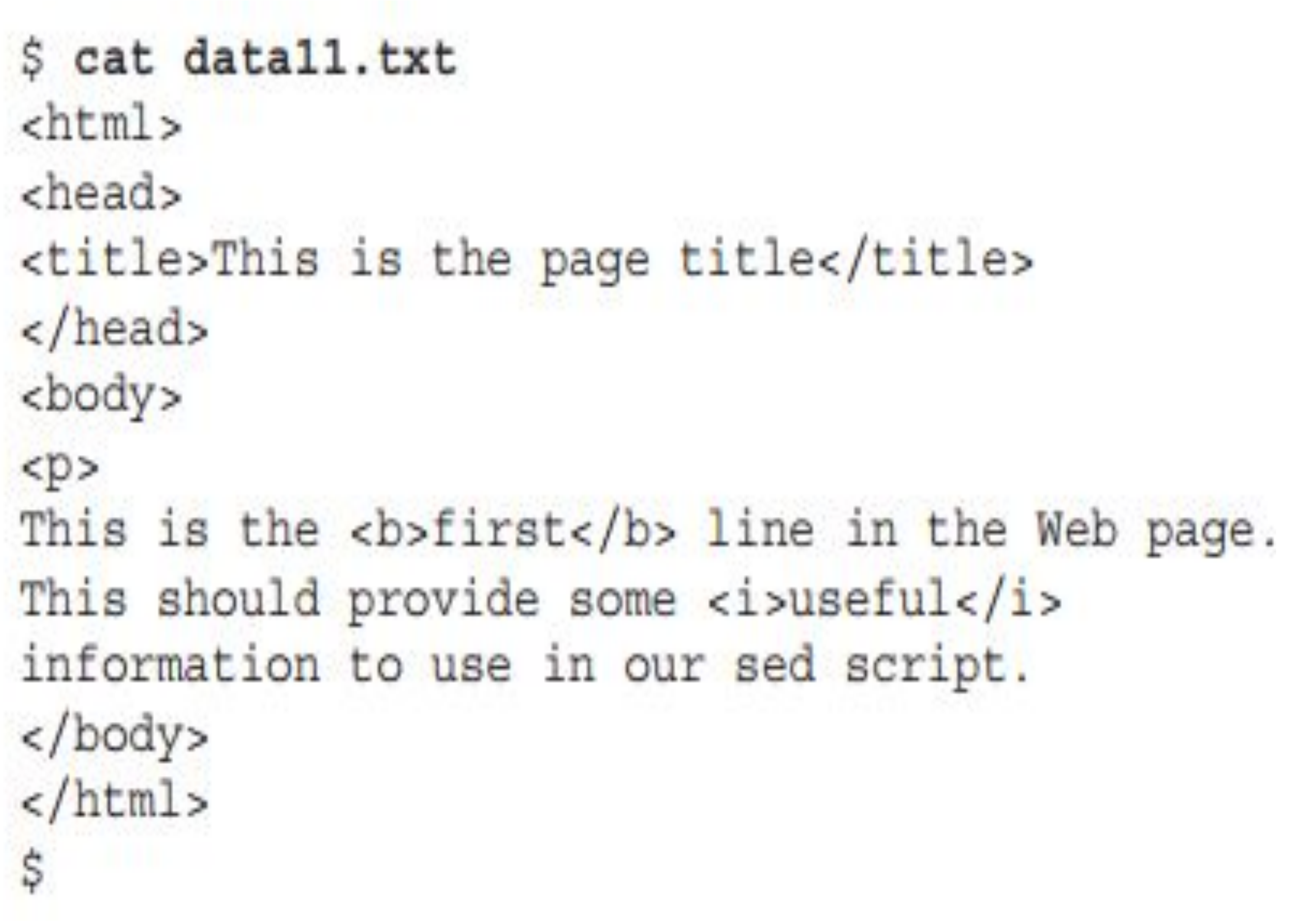
  The script uses an address range to determine what lines are deleted. The range starts with a line that contains a character and continues to the end of the data stream. Any line within this range is not deleted from the output. This means that any lines before the first line that contain a character are deleted.



Look at this simple script in action:

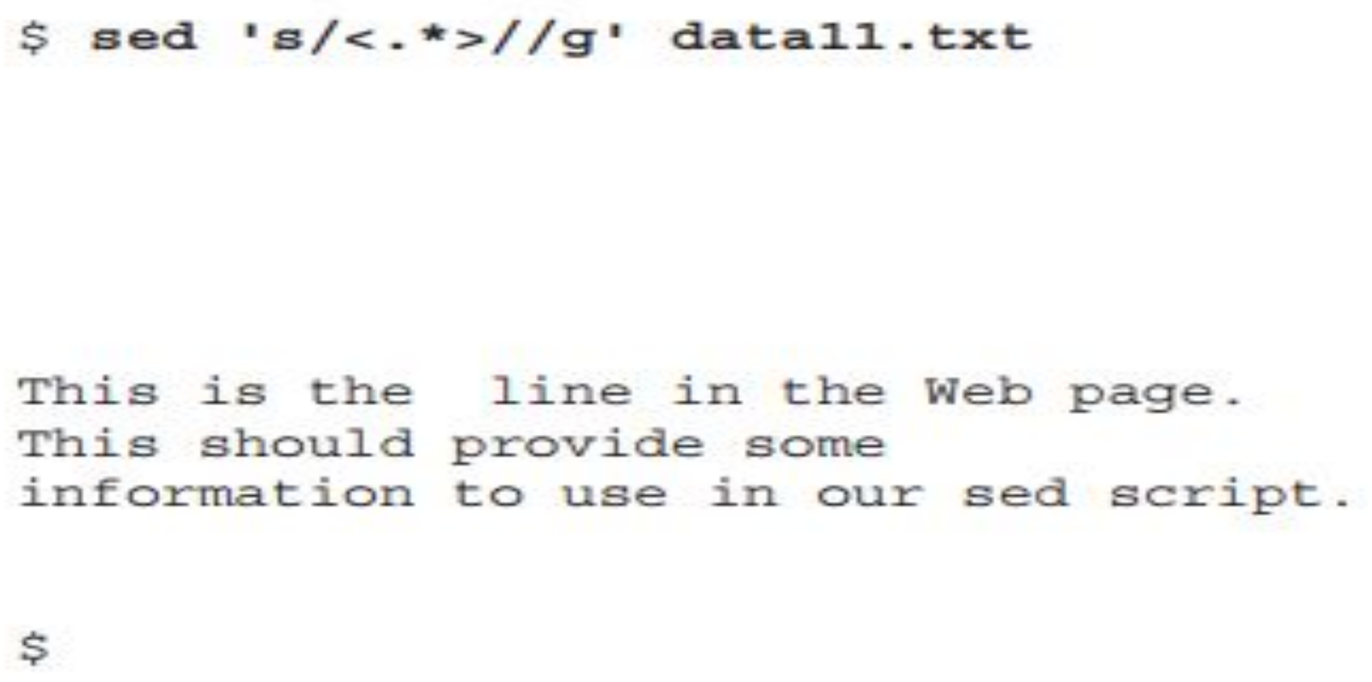
```
$ cat data9.txt


This is line one.

This is line two.
$
$ sed '/./,$!d' data9.txt
This is line one.

This is line two.
$
```

- **Removing HTML tags :**
- A standard HTML web page contains several different types of HTML tags, identifying formatting features required to properly display the page information. Here’s a sample of what an HTML file looks like:



```
$ cat data1.txt
<html>
<head>
<title>This is the page title</title>
</head>
<body>
<p>
This is the <b>first</b> line in the Web page.
This should provide some <i>useful</i>
information to use in our sed script.
</body>
</html>
$
```

- Removing HTML tags creates a problem, however, if you’re not careful. At first glance, you’d think that the way to remove HTML tags would be to just look for a text string that starts with a less-than symbol (<), ends with a greater-than symbol (>), and has data in between the symbols: **`s/<.*>//g`**

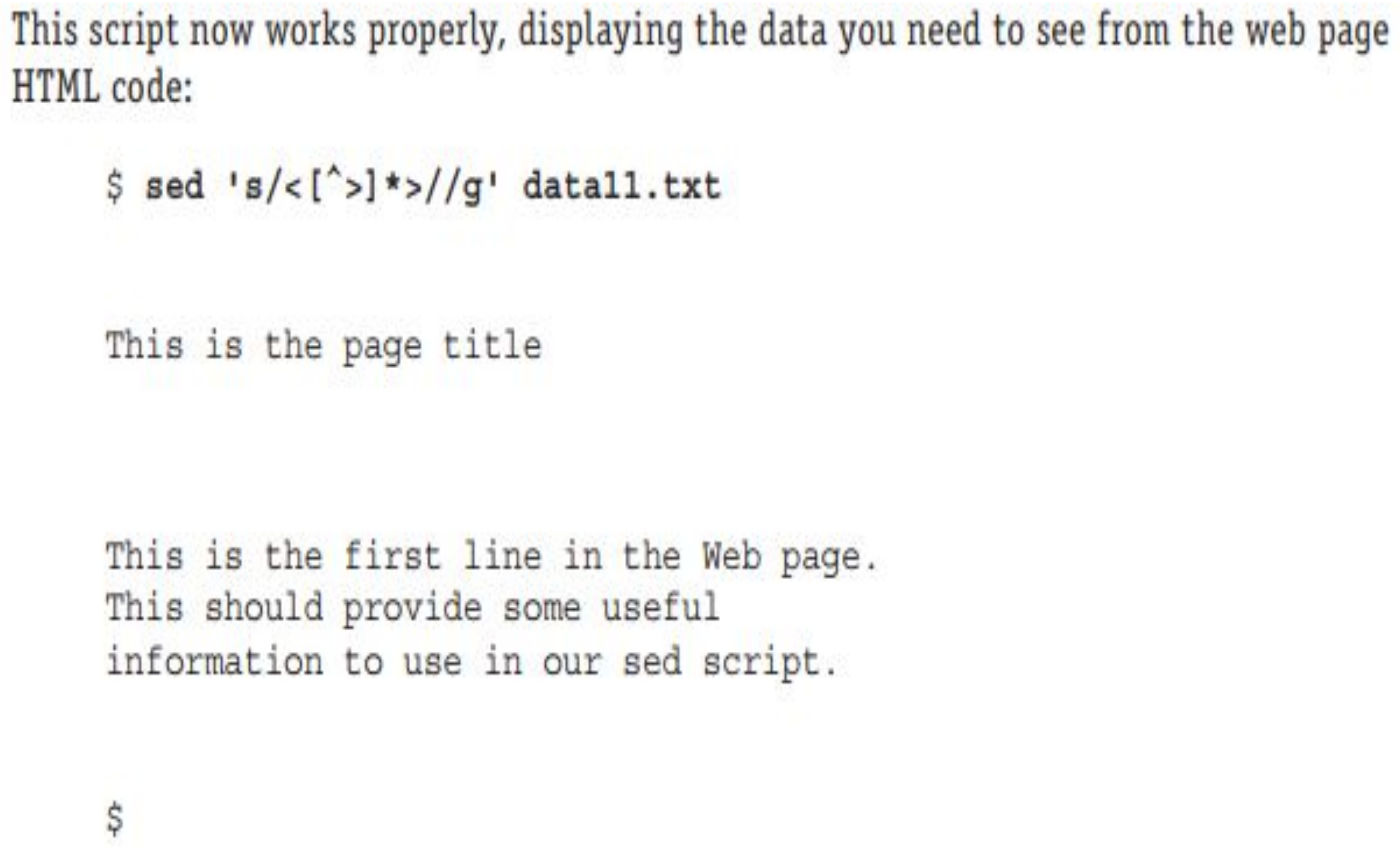


```
$ sed 's/<.*>//g' data11.txt




This is the  line in the Web page.
This should provide some
information to use in our sed script.


$
```

- Notice that the title text is missing, along with the text that was bolded and italicized. The `sed` editor literally interpreted the script to mean any text between the less-than and greater-than sign, including other less-than and greater-than signs! Each time the text was enclosed in HTML tags (such as `<b>first</b>`), the `sed` script removed the entire text.

- The solution to this problem is to have the `sed` editor ignore any embedded greater-than signs between the original tags. To do that, you can create a character class that negates the greater-than sign. This changes the script to:

`s/<[^>]*>//g`


This script now works properly, displaying the data you need to see from the web page HTML code:

```
$ sed 's/<[^>]*>//g' data11.txt


This is the page title



This is the first line in the Web page.
This should provide some useful
information to use in our sed script.


$
```

That's a little better. To clean things up some, you can add a `delete` command to get rid of those pesky blank lines:

```
$ sed 's/<[^>]*>//g ; /^$/d' data11.txt
This is the page title
This is the first line in the Web page.
This should provide some useful
information to use in our sed script.
$
```

Now that's much more compact; there's only the data you need to see.

Advanced gawk

- **Using Variables:**One important feature of any programming language is the ability to store and recall values using variables. The `gawk` programming language supports two different types of variables:
  ■Built-in variables
  ■ User-defined variables
  Several built-in variables are available for you to use in `gawk`. The built-in variables contain information used in handling the data fields and records in the data file. You can also create your own variables in your `gawk` programs.
  Built-in variables:
  The field and record separator variables
  The data field variables allow you to reference individual data fields within a data record using a dollar sign and the numerical position of the data field in the record. Thus, to reference the first data field in the record, you use the `$1` variable. To reference the second data field, you use the `$2` variable, and so on.

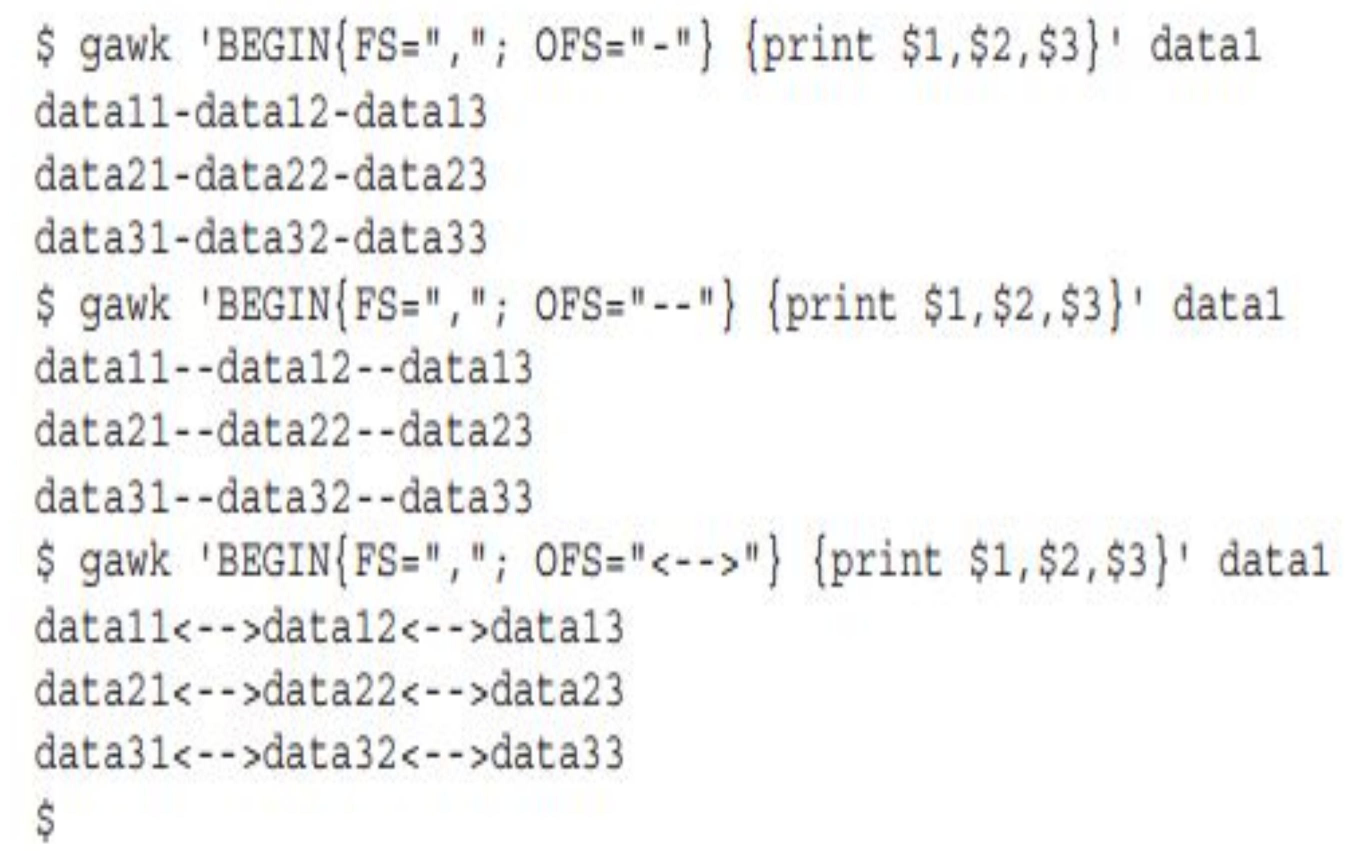
TABLE 22-1 **The gawk Data Field and Record Variables**

Variable	Description
`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

You can see this in the following example:

```
$ cat data1
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
$ gawk 'BEGIN{FS=","} {print $1,$2,$3}' data1
data11 data12 data13
data21 data22 data23
data31 data32 data33
$
```

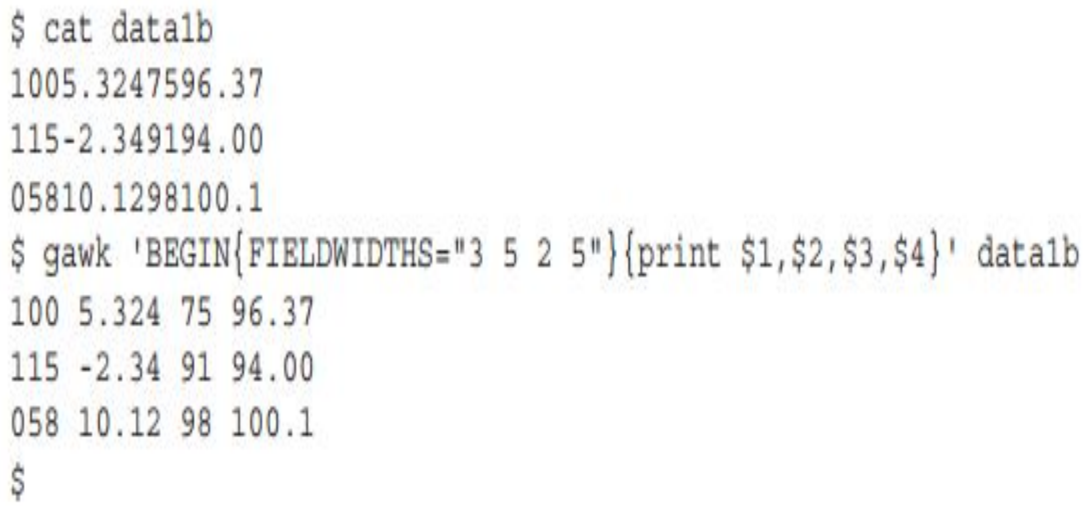
- The `print` command automatically places the value of the `OFS` variable between each data field in the output. By setting the `OFS` variable, you can use any string to separate data fields in the output:



```
$ gawk 'BEGIN{FS=","; OFS="-"} {print $1,$2,$3}' data1
data11-data12-data13
data21-data22-data23
data31-data32-data33
$ gawk 'BEGIN{FS=","; OFS="--"} {print $1,$2,$3}' data1
data11--data12--data13
data21--data22--data23
data31--data32--data33
$ gawk 'BEGIN{FS=","; OFS="<-->"} {print $1,$2,$3}' data1
data11<-->data12<-->data13
data21<-->data22<-->data23
data31<-->data32<-->data33
$
```

- The `FIELDWIDTHS` variable allows you to read records without using a field separator character. In some applications, instead of using a field separator character, data is placed in specific columns within the record. In these instances, you must set the `FIELDWIDTHS` variable to match the layout of the data in the records.

- After you set the `FIELDWIDTHS` variable, `gawk` ignores the `FS` and calculates data fields based on the provided field width sizes. Here's an example using field widths instead of field separator characters:



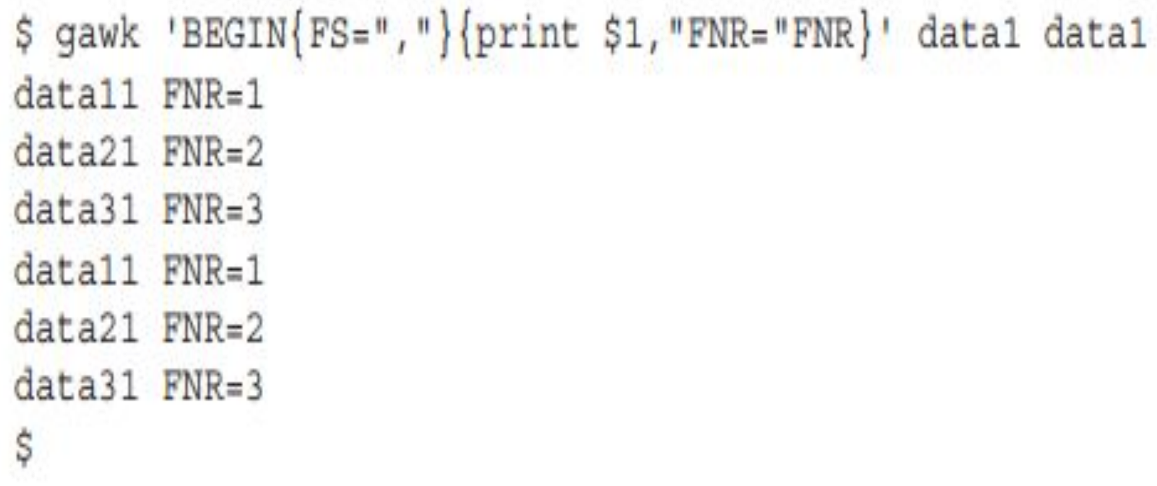
```
$ cat data1b
1005.3247596.37
115-2.349194.00
05810.1298100.1
$ gawk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1,$2,$3,$4}' data1b
100 5.324 75 96.37
115 -2.34 91 94.00
058 10.12 98 100.1
$
```

TABLE 22-2 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 `printf` statement), with a default value of `%.6 g`
`ENVIRON`	An associative array of the current shell environment variables and their values
`ERRNO`	The system error if an error occurs when 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, ignores the case of characters in strings used in the `gawk` command
`NF`	The total number of data fields in the data file
`NR`	The number of input records processed
`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

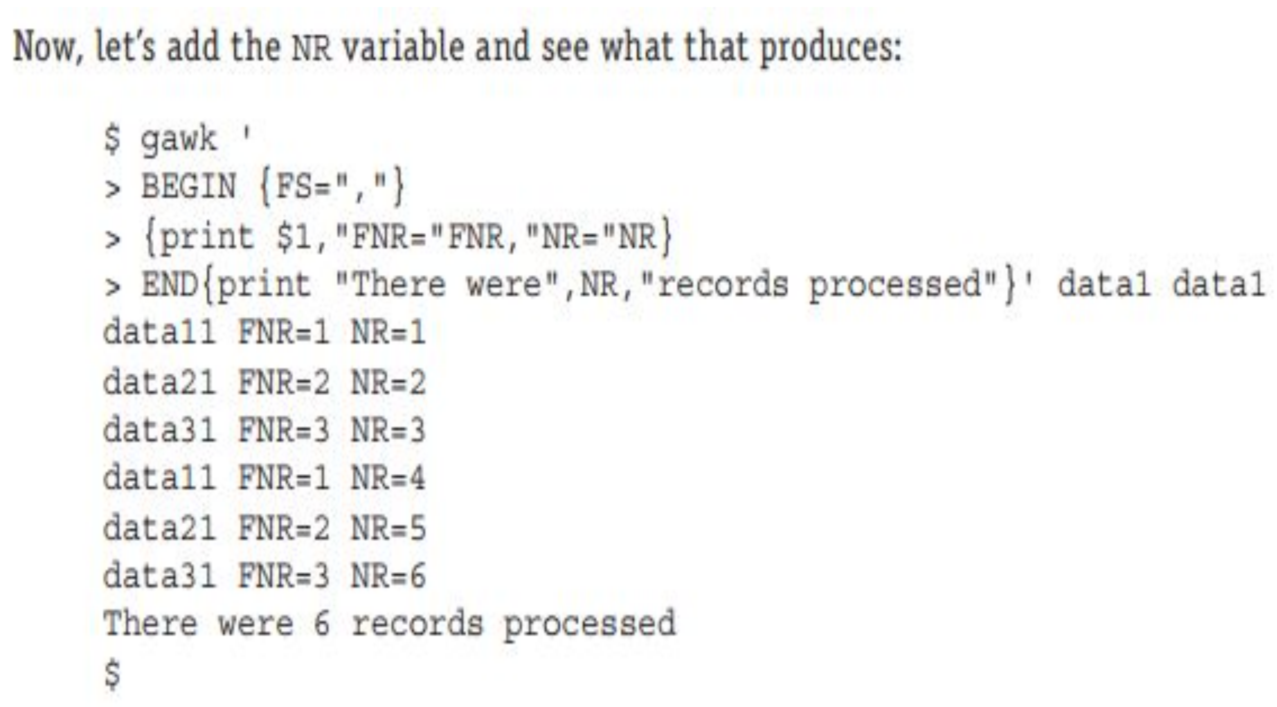
- The `ENVIRON` variable may seem a little odd to you. It uses an *associative array* to retrieve shell environment variables. An associative array uses text for the array index values instead of numeric values.
  The text in the array index is the shell environment variable. The value of the array is the value of the shell environment variable. The following is an example of this:
-
  ```
  $ gawk '
  > BEGIN{
  > print ENVIRON["HOME"]
  > print ENVIRON["PATH"]
  > }'
  /home/rich
  /usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin

  $
  ```
  The `ENVIRON["HOME"]` variable retrieves the `HOME` environment variable value from the shell. Likewise, the `ENVIRON["PATH"]` variable retrieves the `PATH` environment variable value. You can use this technique to retrieve any environment variable value from the shell to use in your `gawk` programs.

- The `FNR` and `NR` variables are similar to each other, but slightly different. **The `FNR` variable contains the number of records processed in the current data file. The `NR` variable contains the total number of records processed.** Let’s look at a couple of examples to see this difference:



```
$ gawk 'BEGIN{FS=","}{print $1,"FNR="FNR}' data1 data1
data11 FNR=1
data21 FNR=2
data31 FNR=3
data11 FNR=1
data21 FNR=2
data31 FNR=3
$
```

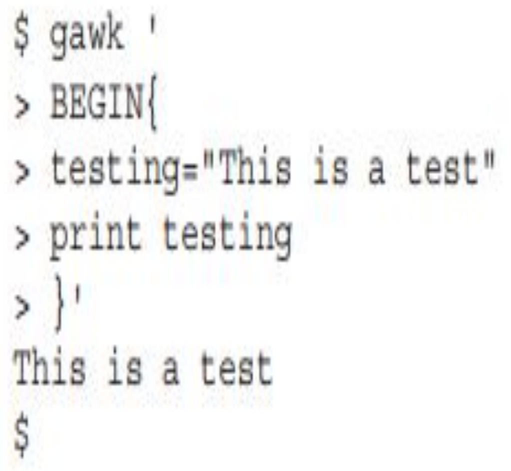
- In this example, the `gawk` program command line defines two input files. (It specifies the same input file twice.) The script prints the first data field value and the current value of the `FNR` variable. Notice that the FNR value was reset to 1 when the gawk program processed the second data file.



Now, let's add the NR variable and see what that produces:

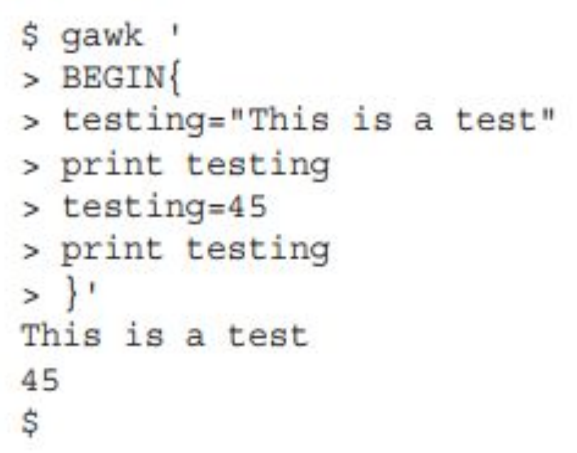
```
$ gawk '
> BEGIN {FS=","}
> {print $1,"FNR="FNR,"NR="NR}
> END{print "There were",NR,"records processed"}' data1 data1
data11 FNR=1 NR=1
data21 FNR=2 NR=2
data31 FNR=3 NR=3
data11 FNR=1 NR=4
data21 FNR=2 NR=5
data31 FNR=3 NR=6
There were 6 records processed
$
```

- **User-defined variables:**
  A `gawk` user-defined variable name can be any number of letters, digits, and underscores, but it can't begin with a digit. It is also important to remember that `gawk` variable names are case sensitive.
  Assigning variables in scripts
  Assigning values to variables in `gawk` programs is similar to doing so in a shell script, using an *assignment statement:*



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

The output of the `print` statement is the current value of the `testing` variable. Like shell script variables, `gawk` variables can hold either numeric or text values:



```
$ gawk '
> BEGIN{
> testing="This is a test"
> print testing
> testing=45
> print testing
> }'
This is a test
45
$
```

Working with Arrays : Many programming languages provide arrays for storing multiple values in a single variable. The `gawk` programming language provides the array feature using *associative arrays.*

Associative arrays are different from numerical arrays in that the index value can be any text string. You don’t have to use sequential numbers to identify data elements contained in the array.

Instead, an associative array consists of a hodge-podge of strings referencing values. Each index string must be unique and uniquely identifies the data element that’s assigned to it. If you’re familiar with other programming languages, this is the same concept as hash maps or dictionaries.

- **Defining array variables:**

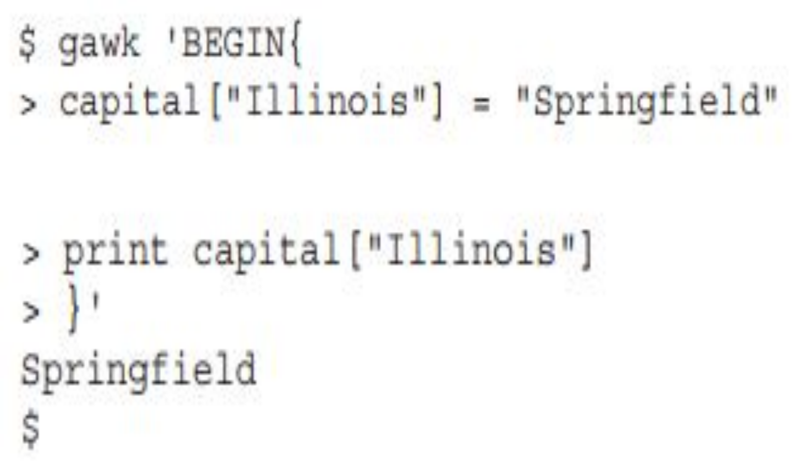
  You can define an array variable using a standard assignment statement. Here's the format of the array variable assignment:

  `var``[`*`index`*`] = `*`element`*

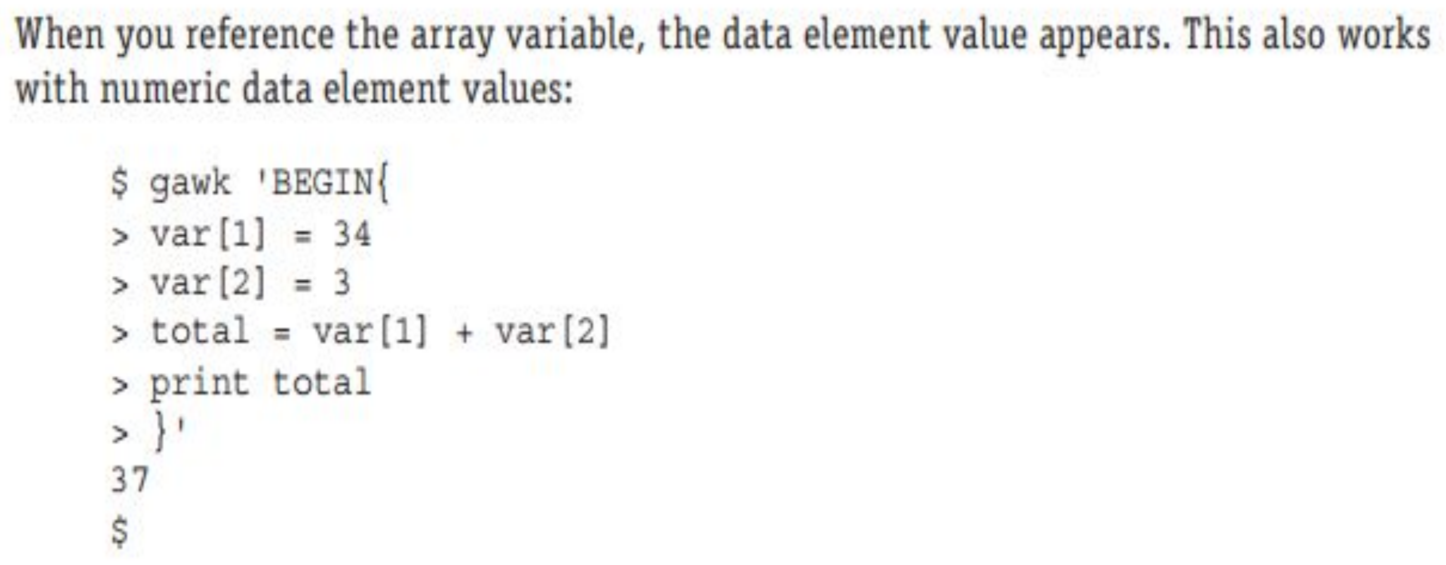
  In this example, *`var`* is the variable name, *`index`* is the associative array index value, and *`element`* is the data element value. Here are some examples of array variables in `gawk`:

-
  ```
  capital["Illinois"] = "Springfield"
  capital["Indiana"] = "Indianapolis"
  capital["Ohio"] = "Columbus"
  ```

  When you reference an array variable, you must include the index value to retrieve the appropriate data element value:



```
$ gawk 'BEGIN{
> capital["Illinois"] = "Springfield"

> print capital["Illinois"]
> }'
Springfield
$
```


When you reference the array variable, the data element value appears. This also works with numeric data element values:

```
$ gawk 'BEGIN{
> var[1] = 34
> var[2] = 3
> total = var[1] + var[2]
> print total
> }'
37
$
```

Iterating through array variables

If you need to iterate through an associate array in `gawk`, you can use a special format of the `for` statement:

```
for (var in array)
{
statements
}
```

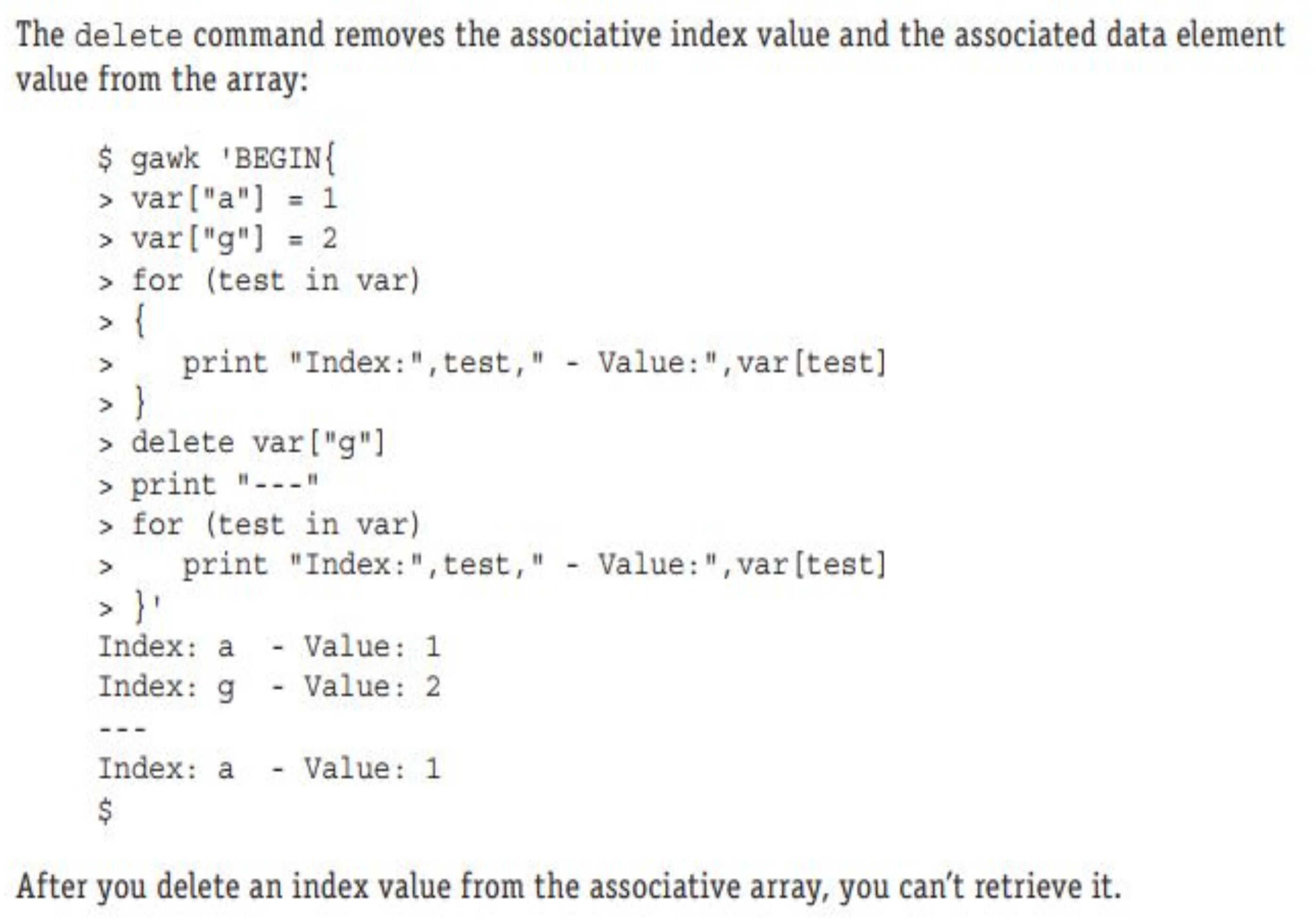
```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> for (test in var)
> {
>    print "Index:",test," - Value:",var[test]
> }
> }'
```

```
Index: u  - Value: 4
Index: m  - Value: 3
Index: a  - Value: 1
Index: g  - Value: 2
$
```

- **Deleting array variables**

  Removing an array index from an associative array requires a special command:

  ```
  delete array[index]
  ```



The `delete` command removes the associative index value and the associated data element value from the array:

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> for (test in var)
> {
>    print "Index:",test," - Value:",var[test]
> }
> delete var["g"]
> print "---"
> for (test in var)
>    print "Index:",test," - Value:",var[test]
> }'
Index: a  - Value: 1
Index: g  - Value: 2
---
Index: a  - Value: 1
$
```

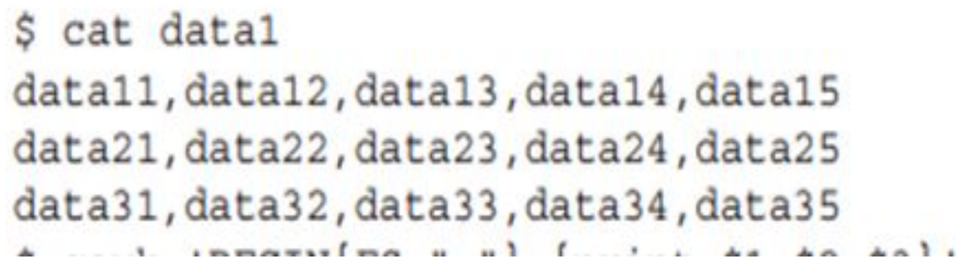
After you delete an index value from the associative array, you can't retrieve it.

- **The matching operator**

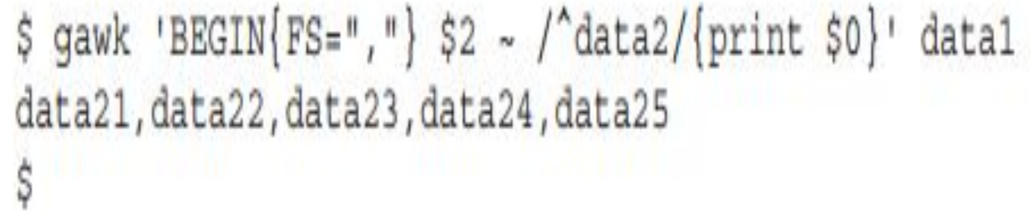
  The *matching operator* allows you to restrict a regular expression to a specific data field in the records. The matching operator is the tilde symbol (~). You specify the matching operator, along with the data field variable, and the regular expression to match:

  `$1 ~ /^data/`

- The `$1` variable represents the first data field in the record. This expression filters records where the first data field starts with the text data. The following is an example of using the matching operator in a `gawk` program script:



```
$ cat data1
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
```



```
$ gawk 'BEGIN{FS=","} $2 ~ /^data2/{print $0}' data1
data21,data22,data23,data24,data25
$
```

- **Structured Commands:**
- **The if statement**
  The `gawk` programming language supports the standard `if-then-else` format of the `if` statement. You must define a condition for the `if` statement to evaluate, enclosed in parentheses. If the condition evaluates to a `TRUE` condition, the statement immediately following the `if` statement is executed. If the condition evaluates to a `FALSE` condition, the statement is skipped. You can use this format:

- `if (`*`condition`*`)`
  `statement1`
  Or you can place it on one line, like this:
  `if (`*`condition`*`)` *`statement1`*

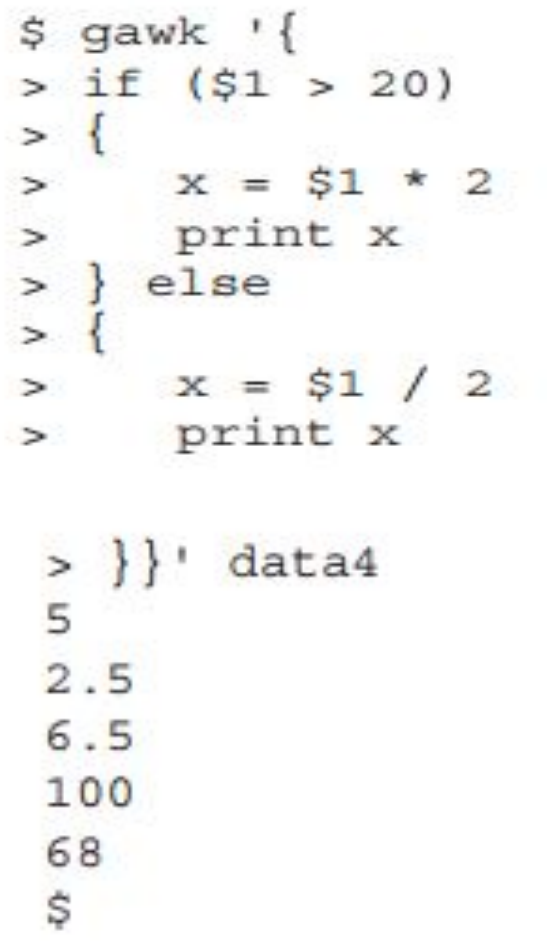
- Here's a simple example demonstrating this format:

```
$ cat data4
10
5
13
50
34
$ gawk '{if ($1 > 20) print $1}' data4
50
34
$
```

Not too complicated. If you need to execute multiple statements in the `if` statement, you must enclose them with braces:

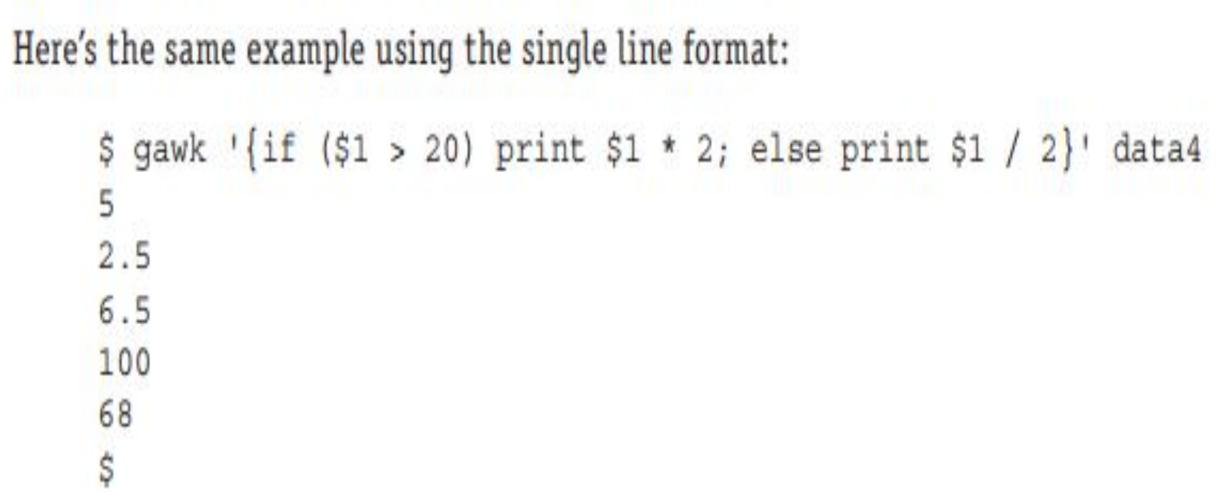
```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }
> }' data4
100
68
$
```

- The `gawk if` statement also supports the `else` clause, allowing you to execute one or more statements if the `if` statement condition fails. Here’s an example of using the `else` clause:



```
$ gawk '{
> if ($1 > 20)
> {
>    x = $1 * 2
>    print x
> } else
> {
>    x = $1 / 2
>    print x

> }}' data4
5
2.5
6.5
100
68
$
```

- You can use the `else` clause on a single line, but you must use a semicolon after the `if` statement section:
  `if (`*`condition`*`)` *`statement1`*`; else` *`statement2`*
  Here's the same example using the single line format:



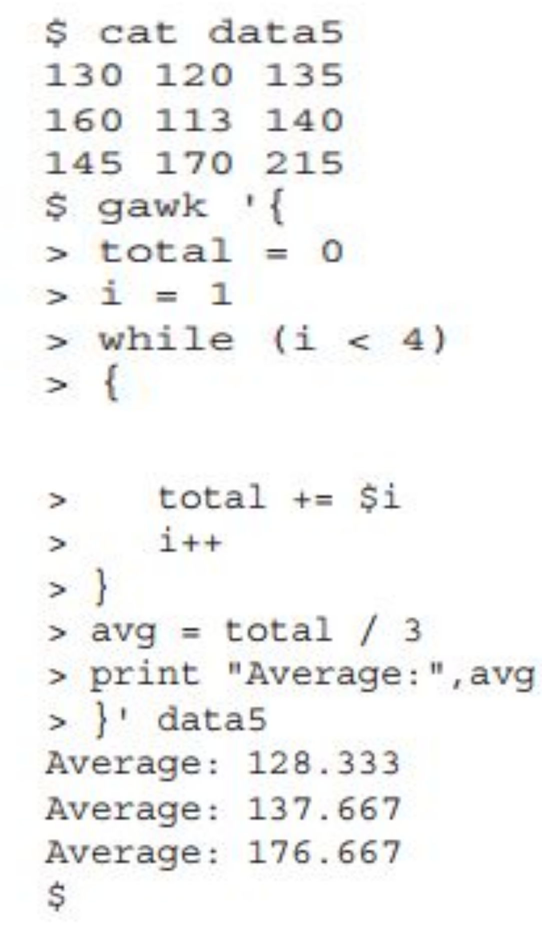
Here's the same example using the single line format:

```
$ gawk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' data4
5
2.5
6.5
100
68
$
```

- **The while statement**

  The `while` statement provides a basic looping feature for `gawk` programs. Here's the format of the `while` statement:

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



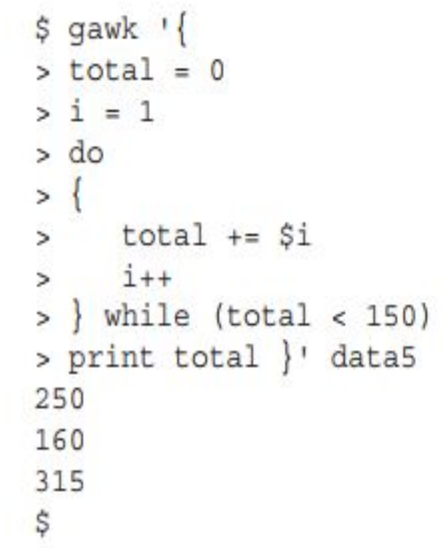
```
$ cat data5
130 120 135
160 113 140
145 170 215
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>    total += $i
>    i++
> }
> avg = total / 3
> print "Average:",avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$
```

The gawk programming language supports using the break and continue statements in while loops, allowing you to jump out of the middle of the loop:

```
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>    total += $i
>    if (i == 2)
>       break
>    i++
> }
> avg = total / 2
> print "The average of the first two data elements is:",avg
> }' data5
The average of the first two data elements is: 125
The average of the first two data elements is: 136.5
The average of the first two data elements is: 157.5
$
```

- **The do-while statement**

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



```
$ gawk '{
> total = 0
> i = 1
> do
> {
>    total += $i
>    i++
> } while (total < 150)
> print total }' data5
250
160
315
$
```

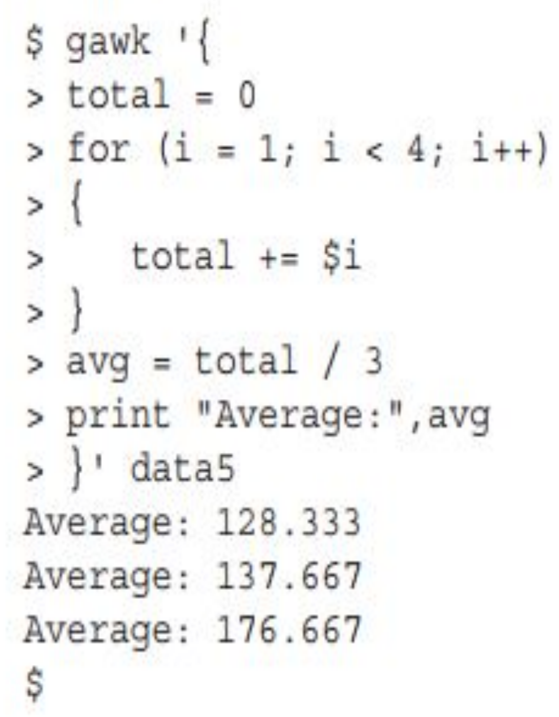
- The script reads the data fields from each record and totals them until the cumulative value reaches 150. If the first data field is over 150 (as seen in the second record), the script is guaranteed to read at least the first data field before evaluating the condition.

- **The for statement :**

The `for` statement is a common method used in many programming languages for looping. The `gawk` programming language supports the C-style of `for` loops:

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

This helps simplify the loop by combining several functions in one statement:



```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>    total += $i
> }
> avg = total / 3
> print "Average:",avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$
```

- **Formatted Printing:** The solution is to use the formatted printing command, called `printf`. If you're familiar with C programming, the `printf` command in `gawk` performs the same way, allowing you
  to specify detailed instructions on how to display data. Here's the format of the `printf` command:

```
printf "format string", var1, var2 . . .
```

  The *format string* is the key to the formatted output. It specifies exactly how the formatted output should appear, using both text elements and *format specifiers.* A format specifier is a special code that indicates what type of variable is displayed and how to display it.
  In addition to the control letters, you can use three modifiers for even more control over your output:

- ■ `width`: This is a numeric value that specifies the minimum width of the output field. If the output is shorter, `printf` pads the space with spaces, using right justification for the text. If the output is longer than the specified width, it overrides the `width` value.

- ■ `prec`: This is a numeric value that specifies the number of digits to the right of the decimal place in floating-point numbers, or the maximum number of characters displayed in a text string.

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>    total += $i
> }
> avg = total / 3
> printf "Average: %5.1f\n",avg
> }' data5
```

```
Average: 128.3
Average: 137.7
Average: 176.7
$
```

• Built-In Functions:

Mathematical functions

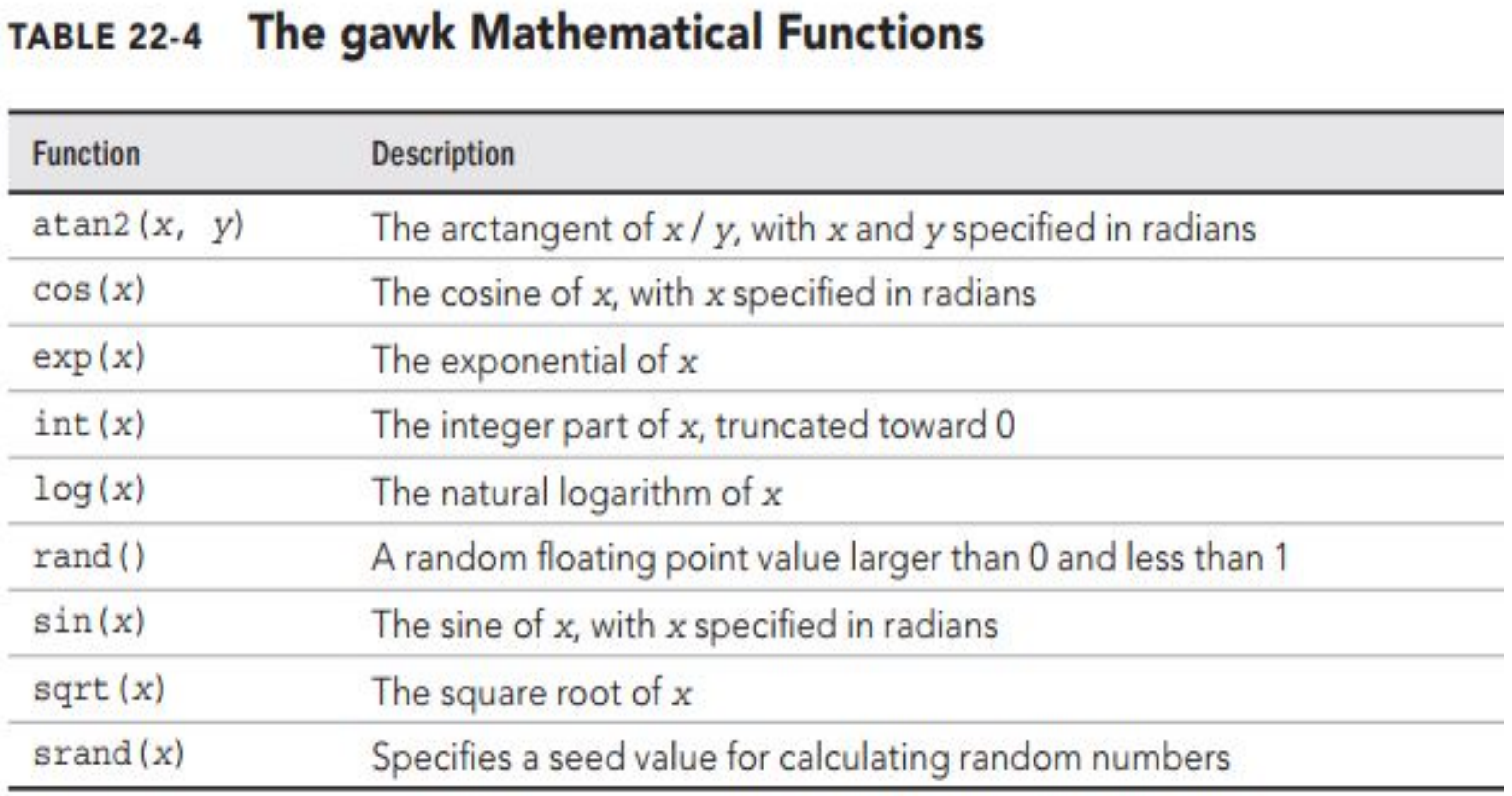


TABLE 22-4 The gawk Mathematical Functions

Function	Description
`atan2(x, y)`	The arctangent of *x* / *y*, with *x* and *y* specified in radians
`cos(x)`	The cosine of *x*, with *x* specified in radians
`exp(x)`	The exponential of *x*
`int(x)`	The integer part of *x*, truncated toward 0
`log(x)`	The natural logarithm of *x*
`rand()`	A random floating point value larger than 0 and less than 1
`sin(x)`	The sine of *x*, with *x* specified in radians
`sqrt(x)`	The square root of *x*
`srand(x)`	Specifies a seed value for calculating random numbers

- Besides the standard mathematical functions, `gawk` also provides a few functions for bitwise manipulating of data:
  - `and(`*`v1, v2`*`)`: Performs a bitwise `AND` of values *`v1`* and *`v2`*
  - `compl(`*`val`*`)`: Performs the bitwise complement of *`val`*
  - `lshift(`*`val, count`*`)`: Shifts the value *`val`* count number of bits left
  - `or(`*`v1, v2`*`)`: Performs a bitwise `OR` of values *`v1`* and *`v2`*
  - `rshift(`*`val, count`*`)`: Shifts the value *`val`* count number of bits right
  - `xor(`*`v1, v2`*`)`: Performs a bitwise `XOR` of values *`v1`* and *`v2`*

  The bit manipulation functions are useful when working with binary values in your data.

- **String functions**

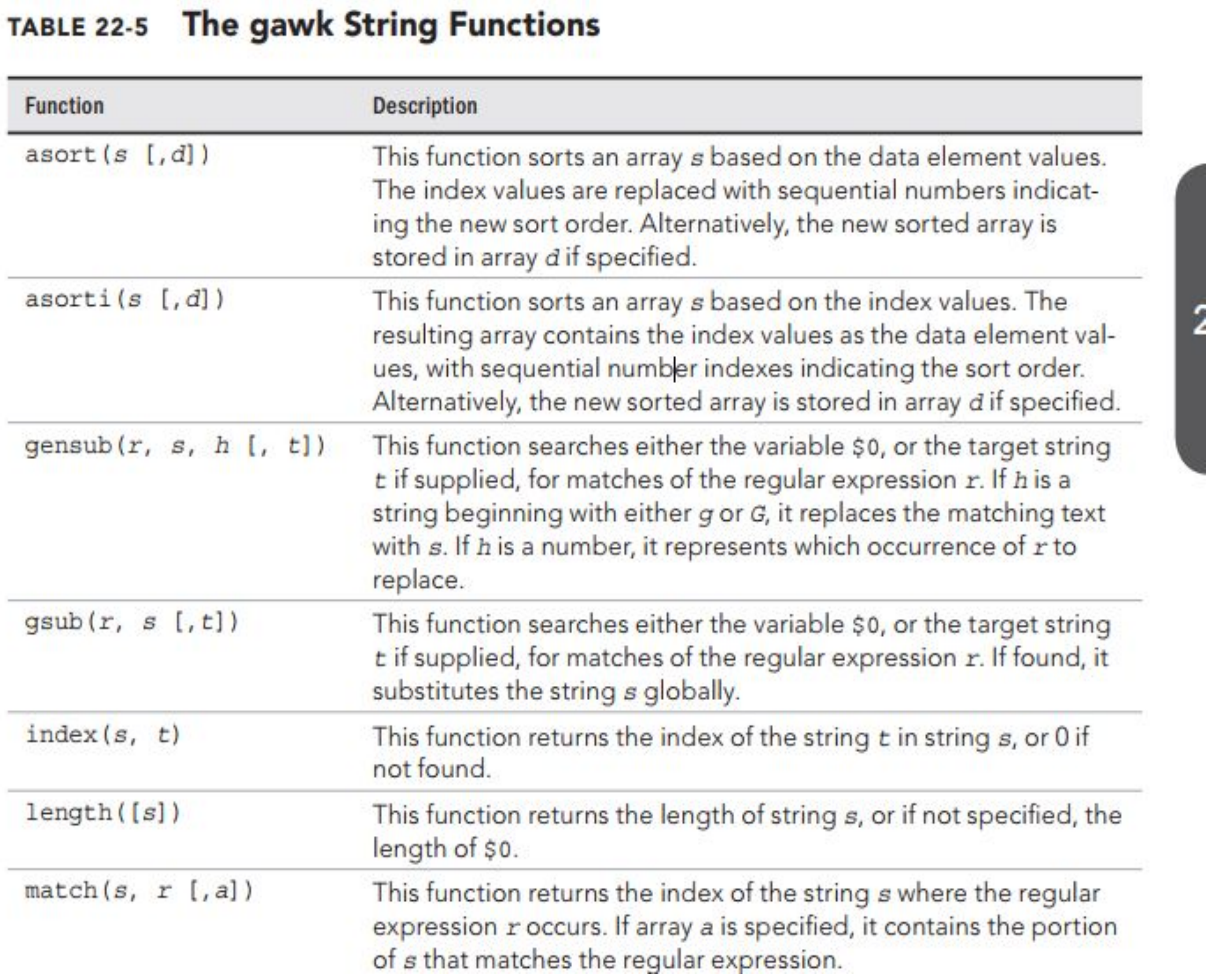
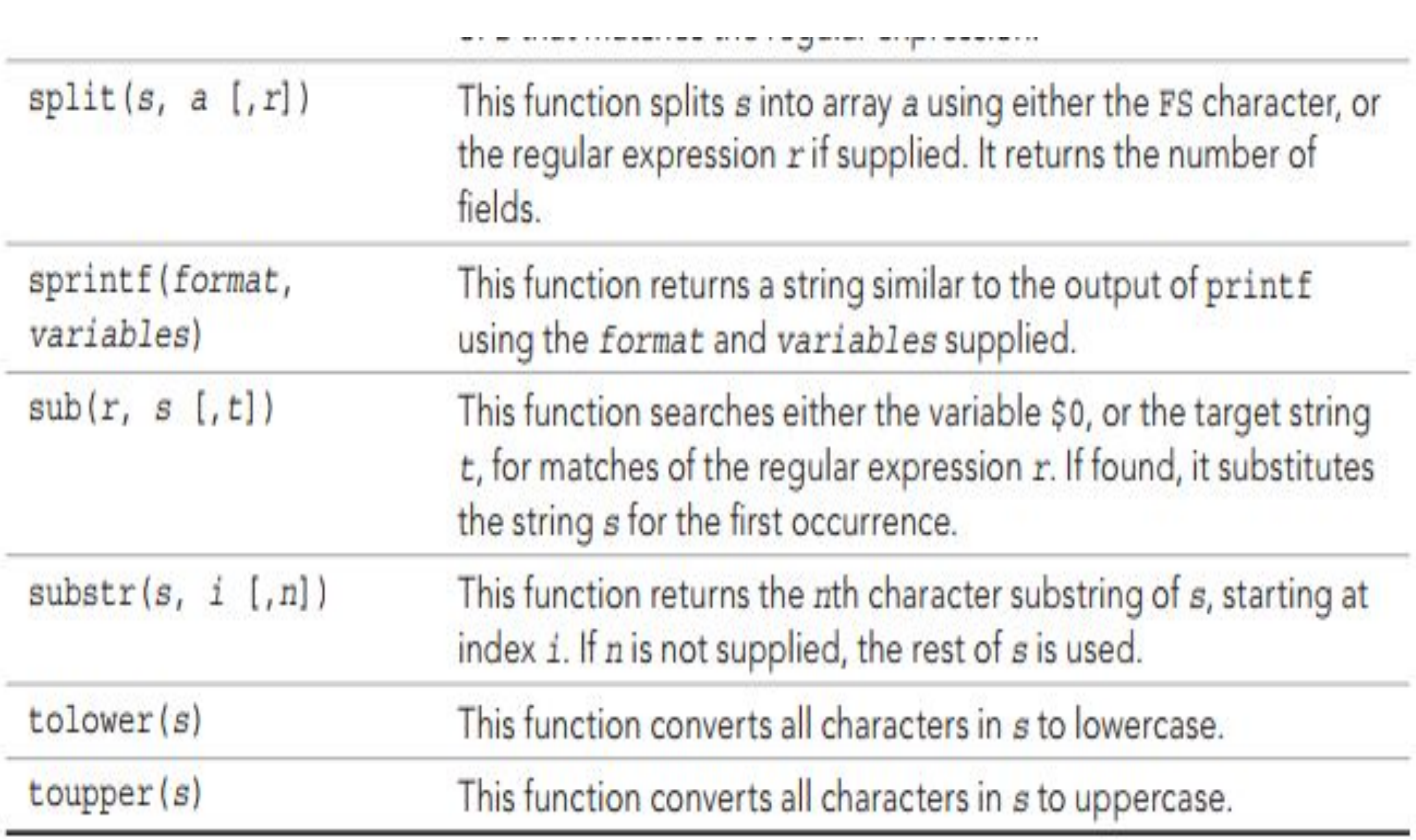


TABLE 22-5 The gawk String Functions

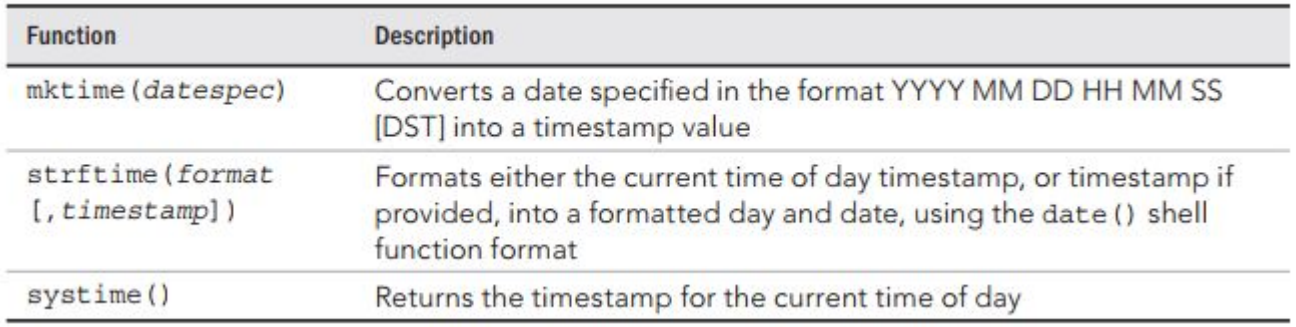
Function	Description
`asort(`*s* `[,`*d*`])`	This function sorts an array *s* based on the data element values. The index values are replaced with sequential numbers indicating the new sort order. Alternatively, the new sorted array is stored in array *d* if specified.
`asorti(`*s* `[,`*d*`])`	This function sorts an array *s* based on the index values. The resulting array contains the index values as the data element values, with sequential number indexes indicating the sort order. Alternatively, the new sorted array is stored in array *d* if specified.
`gensub(`*r*`,` *s*`,` *h* `[,` *t*`])`	This function searches either the variable `$0`, or the target string *t* if supplied, for matches of the regular expression *r*. If *h* is a string beginning with either *g* or *G*, it replaces the matching text with *s*. If *h* is a number, it represents which occurrence of *r* to replace.
`gsub(`*r*`,` *s* `[,`*t*`])`	This function searches either the variable `$0`, or the target string *t* if supplied, for matches of the regular expression *r*. If found, it substitutes the string *s* globally.
`index(`*s*`,` *t*`)`	This function returns the index of the string *t* in string *s*, or 0 if not found.
`length([`*s*`])`	This function returns the length of string *s*, or if not specified, the length of `$0`.
`match(`*s*`,` *r* `[,`*a*`])`	This function returns the index of the string *s* where the regular expression *r* occurs. If array *a* is specified, it contains the portion of *s* that matches the regular expression.



	of s that matches the regular expression.
`split(s, a [,r])`	This function splits *s* into array *a* using either the `FS` character, or the regular expression *r* if supplied. It returns the number of fields.
`sprintf(format, variables)`	This function returns a string similar to the output of `printf` using the *format* and *variables* supplied.
`sub(r, s [,t])`	This function searches either the variable `$0`, or the target string *t*, for matches of the regular expression *r*. If found, it substitutes the string *s* for the first occurrence.
`substr(s, i [,n])`	This function returns the *n*th character substring of *s*, starting at index *i*. If *n* is not supplied, the rest of *s* is used.
`tolower(s)`	This function converts all characters in *s* to lowercase.
`toupper(s)`	This function converts all characters in *s* to uppercase.

Time functions

TABLE 22-6 The gawk Time Functions



Function	Description
`mktime(datespec)`	Converts a date specified in the format YYYY MM DD HH MM SS [DST] into a timestamp value
`strftime(format [,timestamp])`	Formats either the current time of day timestamp, or timestamp if provided, into a formatted day and date, using the `date()` shell function format
`systime()`	Returns the timestamp for the current time of day

- **User-Defined Functions**

  To define you own function, you must use the `function` keyword:

```
function name([variables])
{
statements
}
```

The function name must uniquely identify your function. You can pass one or more variables into the function from the calling `gawk` program:

```
function printthird()
{
print $3
}
```

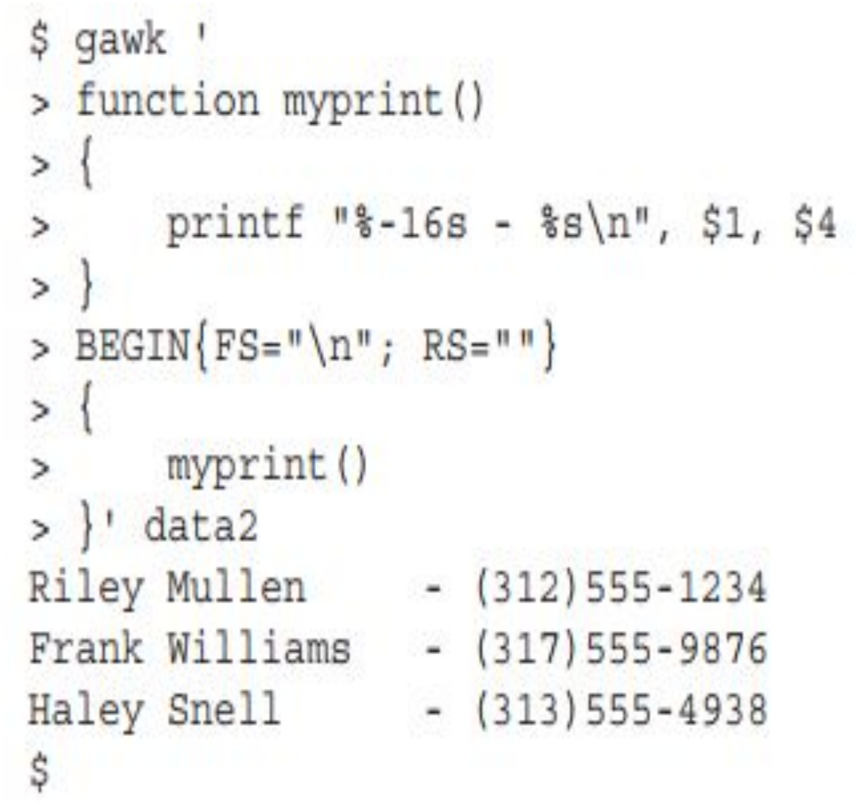
This function prints the third data field in the record. The function can also return a value using the `return` statement:

```
return value
```

The value can be a variable, or an equation that evaluates to a value:

```
function myrand(limit)
{
return int(limit * rand())
}
```

- **Using your functions**
  When you define a function, it must appear by itself before you define any programming sections (including the `BEGIN` section). This may look a little odd at first, but it helps keep the function code separate from the rest of the `gawk` program:



```
$ gawk '
> function myprint()
> {
>     printf "%-16s - %s\n", $1, $4
> }
> BEGIN{FS="\n"; RS=""}
> {
>     myprint()
> }' data2
Riley Mullen     - (312)555-1234
Frank Williams   - (317)555-9876
Haley Snell      - (313)555-4938
$
```