

# Regular Expressions

## IN THIS CHAPTER

Defining regular expressions

Looking at the basics

Extending our patterns

Creating expressions

**T**he key to successfully working with the `sed` editor and the `gawk` program in your shell script is your comfort using regular expressions. This is not always an easy thing to do, because trying to filter specific data from a large batch of data can (and often does) get complicated. This chapter describes how to create regular expressions in both the `sed` editor and the `gawk` program that can filter out just the data you need.

## What Are Regular Expressions?

---

The first step to understanding regular expressions is to define just exactly what they are. This section explains what a regular expression is and describes how Linux uses regular expressions.

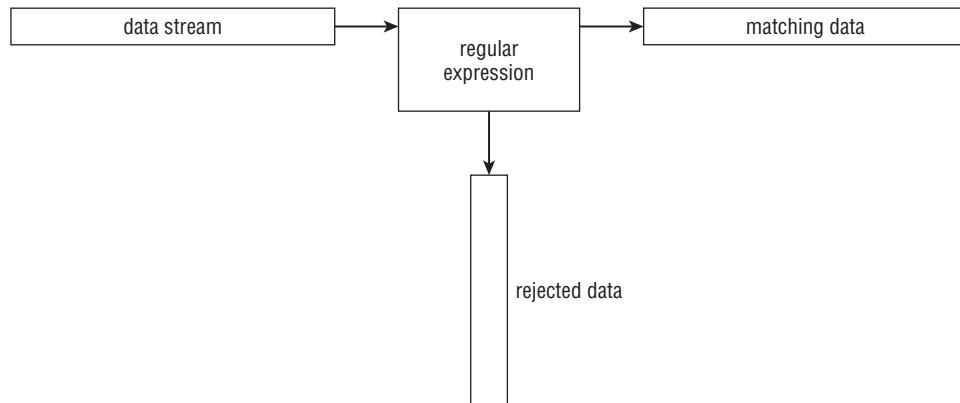
### A definition

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. This is illustrated in Figure 20-1.

The regular expression pattern makes use of wildcard characters to represent one or more characters in the data stream. There are plenty of instances in Linux where you can specify a wildcard character to represent data you don't know about. You've already seen an example of using wildcard characters with the Linux `ls` command for listing files and directories (see Chapter 3).

**FIGURE 20-1**

Matching data against a regular expression pattern



The asterisk wildcard character allows you to list only files that match a certain criteria. For example:

```
$ ls -al da*
-rw-r--r--  1 rich      rich          45 Nov 26 12:42 data
-rw-r--r--  1 rich      rich          25 Dec  4 12:40 data.tst
-rw-r--r--  1 rich      rich        180 Nov 26 12:42 data1
-rw-r--r--  1 rich      rich          45 Nov 26 12:44 data2
-rw-r--r--  1 rich      rich          73 Nov 27 12:31 data3
-rw-r--r--  1 rich      rich          79 Nov 28 14:01 data4
-rw-r--r--  1 rich      rich        187 Dec  4 09:45 datatest
$
```

The `da*` parameter instructs the `ls` command to list only the files whose name starts with *da*. There can be any number of characters after the *da* in the filename (including none). The `ls` command reads the information regarding all the files in the directory but displays only the ones that match the wildcard character.

Regular expression wildcard patterns work in a similar way. The regular expression pattern contains text and/or special characters that define a template for the `sed` editor and the `gawk` program to follow when matching data. You can use different special characters in a regular expression to define a specific pattern for filtering data.

## Types of regular expressions

The biggest problem with using regular expressions is that there isn't just one set of them. Several different applications use different types of regular expressions in the Linux environment. These include such diverse applications as programming languages (Java, Perl, and

Python), Linux utilities (such as the `sed` editor, the `gawk` program, and the `grep` utility), and mainstream applications (such as the MySQL and PostgreSQL database servers).

A regular expression is implemented using a *regular expression engine*. A regular expression engine is the underlying software that interprets regular expression patterns and uses those patterns to match text.

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.

Because there are so many different ways to implement regular expressions, it's hard to present a single, concise description of all the possible regular expressions. The following sections discuss the most commonly found regular expressions and demonstrate how to use them in the `sed` editor and `gawk` program.

## Defining BRE Patterns

The most basic BRE pattern is matching text characters in a data stream. This section demonstrates how you can define text in the regular expression pattern and what to expect from the results.

### Plain text

Chapter 18 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.

You probably already noticed that the regular expression doesn’t care where in the data stream the pattern occurs. It also doesn’t matter how many times the pattern occurs. After the regular expression can match the pattern anywhere in the text string, it passes the string along to the Linux utility that’s using it.

The key is matching the regular expression pattern to the data stream text. It’s important to remember that regular expressions are extremely picky about matching patterns. 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.

You don’t have to limit yourself to whole words in the regular expression. If the defined text appears anywhere in the data stream, the regular expression matches the following:

```
$ echo "The books are expensive" | sed -n '/book/p'
The books are expensive
$
```

Even though the text in the data stream is *books*, the data in the stream contains the regular expression *book*, so the regular expression pattern matches the data. Of course, if you try the opposite, the regular expression fails:

```
$ echo "The book is expensive" | sed -n '/books/p'
$
```

The complete regular expression text didn't appear in the data stream, so the match failed and the `sed` editor didn't display the text.

You also don't have to limit yourself to single text words in the regular expression. You can include spaces and numbers in your text string as well:

```
$ echo "This is line number 1" | sed -n '/ber 1/p'
This is line number 1
$
```

Spaces are treated just like any other character in the regular expression:

```
$ echo "This is line number1" | sed -n '/ber 1/p'
$
```

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.
$
```

The line with two spaces between words matches the regular expression pattern. This is a great way to catch spacing problems in text files!

## Special characters

As you use text strings in your regular expression patterns, there's something you need to be aware of. There are a few exceptions when defining text characters in a regular expression. 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:

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

As the chapter progresses, you'll find out just what these special characters do in a regular expression. For now, however, just remember that you can't use these characters by themselves in your text pattern.

If you want to use one of the special characters as a text character, you need to *escape* it. When you escape the special characters, you add a special character in front of it to indicate to the regular expression engine that it should interpret the next character as a normal text character. The special character that does this is the backslash character (`\`).

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
$
```

Finally, although the forward slash isn't a regular expression special character, if you use it in your regular expression pattern in the `sed` editor or the `gawk` program, you get an error:

```
$ echo "3 / 2" | sed -n '///p'
sed: -e expression #1, char 2: No previous regular expression
$
```

To use a forward slash, you need to escape that as well:

```
$ echo "3 / 2" | sed -n '/\\/p'
3 / 2
$
```

Now the `sed` editor can properly interpret the regular expression pattern, and all is well.

## Anchor characters

As shown in the “Plain Text” section, by default, when you specify a regular expression pattern, if the pattern appears anywhere in the data stream, it matches. You can use two special characters to anchor a pattern to either the beginning or the end of lines in the data stream.

### 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
$
```

The caret anchor character checks for the pattern at the beginning of each new line of data, as determined by the newline character:

```
$ cat data3
This is a test line.
this is another test line.
A line that tests this feature.
Yet more testing of this
$ sed -n '/^this/p' data3
this is another test line.
$
```

As long as the pattern appears at the start of a new line, the caret anchor catches it.

If you position the caret character in any place other than at the beginning of the pattern, it acts like a normal character and not as a special character:

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

Because the caret character is listed last in the regular expression pattern, the `sed` editor uses it as a normal character to match text.

## NOTE

If you need to specify a regular expression pattern using only the caret character, you don't need to escape it with a backslash. However, if you specify the caret character first, followed by additional text in the pattern, you need to use the escape character before the caret character.

## 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'
$
```

The problem with an ending text pattern is that you must be careful what you're looking for:

```
$ echo "There are a lot of good books" | sed -n '/book$/p'
$
```

Making the word “book” plural at the end of the line means that it no longer matches the regular expression pattern, even though book is in the data stream. The text pattern must be the last thing on the line for the pattern to match.

### 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 `sed` editor ignores the lines that include other text besides the specified text.

The second situation may seem a little odd at first but is extremely useful. By combining both anchors in a pattern with no text, you can filter blank lines from the data stream. Consider this example:

```
$ cat data5
This is one test line.

This is another test line.
$ sed '/^$/d' data5
This is one test line.
This is another test line.
$
```

The regular expression pattern that is defined looks for lines that have nothing between the start and end of the line. Because blank lines contain no text between the two newline characters, they match the regular expression pattern. The `sed` editor uses the `d` delete command to delete lines that match the regular expression pattern, thus removing all blank lines from the text. This is an effective way to remove blank lines from documents.

### 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.

Let's look at a few examples of using the dot character in a regular expression pattern:



```
$ 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. The fifth line proves this, by putting the `at` in the front of the line, which fails to match the pattern.

## Character classes

The dot special character is great for matching a character position against any character, but what if you want to limit what characters to match? This is called a *character class* in regular expressions.

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. This takes a little getting used to at first, but after you catch on, it can generate some pretty amazing results.

The following is an example of creating a character class:

```
$ sed -n '/[ch]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

Using the same data file as in the dot special character example, we came up with a different result. This time we managed to filter out the line that just contained the word `at`. The only words that match this pattern are `cat` and `hat`. Also notice that the line that started with `at` didn't match as well. There must be a character in the character class that matches the appropriate position.

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
$
```

The regular expression used three character classes to cover both lower and upper cases for all three character positions.

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.

You can combine character classes to check for properly formatted numbers, such as phone numbers and ZIP codes. However, when you're trying to match a specific format, you must be careful. Here's an example of a ZIP code match gone wrong:

```
$ cat data8
60633
46201
223001
4353
22203
$ sed -n '
>/[0123456789][0123456789][0123456789][0123456789][0123456789]/p
>' data8
```

```
60633
46201
223001
22203
$
```

This might not have produced the result you were thinking of. It did a fine job of filtering out the number that was too short to be a ZIP code, because the last character class didn't have a character to match against. However, it still passed the six-digit number, even though we only defined five character classes.

Remember that the regular expression pattern can be found anywhere in the text of the data stream. You may always have additional characters besides the matching pattern characters. If you want to ensure that you match against only five numbers, you need to delineate them somehow, either with spaces, or as in this example, by showing that they're at the start and end of the line:

```
$ sed -n '
> /^[0123456789][0123456789][0123456789][0123456789][0123456789]$/p
> ' data8
60633
46201
22203
$
```

Now that's much better! Later in this chapter, we look at how to simplify this even further.

One extremely popular use for character classes is parsing words that might be misspelled, such as data entered from a user form. You can easily create regular expressions that can accept common misspellings in data:

```
$ cat data9
I need to have some maintenance done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$ sed -n '
/maint[ea]n[ae]nce/p
/sep[ea]r[ea]te/p
' data9
I need to have some maintenance done on my car.
I'll pay that in a seperate invoice.
After I pay for the maintenance my car will be as good as new.
$
```

The two `sed` print commands in this example utilize regular expression character classes to help catch the misspelled words, *maintenance* and *separate*, in the text. The same regular expression pattern also matches the properly spelled occurrence of “maintenance.”

### Negating character classes

In regular expression patterns, you can also reverse the effect of a character class. Instead of looking for a character contained in the class, you can look for any character that's not in the class. To do that, just place a caret character at the beginning of the character class range:

```
$ 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.

### Using ranges

You may have noticed when I showed the ZIP code example earlier that it was somewhat awkward having to list all the possible digits in each character class. Fortunately, you can use a shortcut so you don't have to do that.

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 (see Chapter 2).

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'
$
```

The same technique works with letters:

```
$ sed -n '/[c-h]at/p' data6
The cat is sleeping.
```

```
That is a very nice hat.
$
```

The new pattern `[c-h]at` matches words where the first letter is between the letter *c* and the letter *h*. In this case, the line with only the word `at` failed to match the pattern.

You can also specify multiple, non-continuous ranges in a single character class:

```
$ sed -n '/[a-ch-m]at/p' data6
The cat is sleeping.
That is a very nice hat.
$
```

The character class allows the ranges *a* through *c*, and *h* through *m* to appear before the `at` text. This range would reject any letters between *d* and *g*:

```
$ echo "I'm getting too fat." | sed -n '/[a-ch-m]at/p'
$
```

This pattern rejected the `fat` text, as it wasn't in the specified range.

## Special character classes

In addition to defining your own character classes, the BRE contains special character classes you can use to match against specific types of characters. Table 20-1 describes the BRE special characters you can use.

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

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

Using the special character classes is an easy way to define ranges. Instead of having to use a range [0-9], you can just use [[:digit:]].

### 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 "ieeeeek" | sed -n '/ie*k/p'
ieeeeek
$
```

This pattern symbol is commonly used for handling words that have a common misspelling or variations in language spellings. For example, if you need to write a script that may be used in either American or British English, you could write:

```
$ echo "I'm getting a color TV" | sed -n '/colou*r/p'
I'm getting a color TV
$ echo "I'm getting a colour TV" | sed -n '/colou*r/p'
I'm getting a colour TV
$
```

The `u*` in the pattern indicates that the letter `u` may or may not appear in the text to match the pattern. Similarly, if you know of a word that is commonly misspelled, you can accommodate it by using the asterisk:

```
$ echo "I ate a potatoe with my lunch." | sed -n '/potatoe*/p'
I ate a potatoe with my lunch.
$ echo "I ate a potato with my lunch." | sed -n '/potatoe*/p'
I ate a potato with my lunch.
$
```

Placing an asterisk next to the possible extra letter allows you to accept the misspelled word.

Another handy feature is combining the dot special character with the asterisk special character. This combination provides a pattern to match any number of any characters. It's often used between two text strings that may or may not appear next to each other in the data stream:

```
$ echo "this is a regular pattern expression" | sed -n '
> /regular.*expression/p'
this is a regular pattern expression
$
```

Using this pattern, you can easily search for multiple words that may appear anywhere in a line of text in the data stream.

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

As long as the *a* and *e* characters appear in any combination between the *b* and *t* characters (including not appearing at all), the pattern matches. If any other character outside of the defined character class appears, the pattern match fails.

## Extended Regular Expressions

The POSIX ERE patterns include a few additional symbols that are used by some Linux applications and utilities. The `gawk` program recognizes the ERE patterns, but the `sed` editor doesn't.

### CAUTION

Remember that the regular expression engines in the `sed` editor and the `gawk` program are different. The `gawk` program can use most of the extended regular expression pattern symbols, and it can provide some additional filtering capabilities that the `sed` editor doesn't have. However, because of this, it is often slower in processing data streams.

This section describes the more commonly found ERE pattern symbols that you can use in your `gawk` program scripts.

### The question mark

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
$
```

This time if either character defined in the character class appears, the text matches the specified pattern.

## 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.

This feature allows you to fine-tune exactly how many times you allow a character (or character class) to appear in a pattern.

### CAUTION

By default, the `gawk` program doesn't recognize regular expression intervals. You must specify the `--re-interval` command line option for the `gawk` program to recognize regular expression intervals.

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}'
$
```

By specifying an interval of one, you restrict the number of times the character can be present for the string to match the pattern. If the character appears more times, the pattern match fails.

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}'
$
```

In this example, the `e` character can appear once or twice for the pattern match to pass; otherwise, the pattern match fails.

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}'
$
```

```
$ echo "baeaet" | gawk --re-interval '/b[ae]{1,2}t/{print $0}'
$
```

This regular expression pattern matches if there are exactly one or two instances of the letter *a* or *e* in the text pattern, but it fails if there are any more in any combination.

## The pipe symbol

The pipe symbol allows 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}'
$
```

This example looks for the regular expression *cat* or *dog* in the data stream. You can't place any spaces within the regular expressions and the pipe symbol, or they're added to the regular expression pattern.

The regular expressions on either side of the pipe symbol can use any regular expression pattern, including character classes, to define the text:

```
$ echo "He has a hat." | gawk '/[ch]at|dog/{print $0}'
He has a hat.
$
```

This example would match *cat*, *hat*, or *dog* in the data stream text.

## Grouping expressions

Regular expression patterns can also be grouped by using parentheses. When you group a regular expression pattern, the group is treated like a standard character. You can apply a special character to the group just as you would to a regular character. For example:

```
$ echo "Sat" | gawk '/Sat(urday)?/{print $0}'
Sat
$ echo "Saturday" | gawk '/Sat(urday)?/{print $0}'
Saturday
$
```

The grouping of the “urday” ending along with the question mark allows the pattern to match either the full day name Saturday or the abbreviated name Sat.

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.

## Regular Expressions in Action

---

Now that you’ve seen the rules and a few simple demonstrations of using regular expression patterns, it’s time to put that knowledge into action. The following sections demonstrate some common regular expression examples within shell scripts.

### 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. Chapter 6 showed you how to display the `PATH` environment variable:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/
local/games
$
```

Your `PATH` environment variable will differ, depending on where the applications are located on your Linux system. The key is to recognize that each directory in the `PATH` is separated by a colon. To get a listing of directories that you can use in a script, you must replace each colon with a space. You now recognize that the `sed` editor can do just that using a simple regular expression:

```
$ echo $PATH | sed 's:/ /g'
/usr/local/sbin /usr/local/bin /usr/sbin /usr/bin /sbin /bin
/usr/games /usr/local/games
$
```

After you have the directories separated out, you can use them in a standard `for` statement (see Chapter 13) to iterate through each directory:

```
mypath=$(echo $PATH | sed 's:/ /g')
for directory in $mypath
do
...
done
```

After you have each directory, you can use the `ls` command to list each file in each directory, and use another `for` statement to iterate through each file, incrementing a counter for each file.

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
$
```

Now we're starting to see some of the power behind regular expressions!

### Validating a phone number

The previous example showed how to incorporate the simple regular expression along with `sed` to replace characters in a data stream to process data. Often, regular expressions are used to validate data to ensure that data is in the correct format for a script.

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
```

This leaves four possibilities for how customers can enter their phone number in a form. The regular expression must be robust enough to handle any of these situations.

When building a regular expression, it's best to start on the left side and build your pattern to match the possible characters you'll run into. 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.

Next is the three-digit area code. In the United States, area codes start with the number 2 (no area codes start with the digits 0 or 1), and can go to 9. To match the area code, you'd use the following pattern:

```
[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}$
```

You can use this regular expression pattern in the `gawk` program to filter out bad phone numbers. Now you just need to create a simple script using the regular expression in a `gawk` program and filter your phone list through the script. Remember that when you use regular expression intervals in the `gawk` program, you must use the `--re-interval` command line option, or you won't get the correct results.

Here's the script:

```
$ cat isphone
#!/bin/bash
# script to filter out bad phone numbers
gawk --re-interval '/^\(?[2-9][0-9]{2}\)?(| - | \. ) [0-9]{3} ( | - | \. ) [0-9]{4}$/'
$
```

Although you can't tell from this listing, the `gawk` command is on a single line in the shell script. You can then redirect phone numbers to the script for processing:

```
$ echo "317-555-1234" | ./isphone
317-555-1234
$ echo "000-555-1234" | ./isphone
$ echo "312 555-1234" | ./isphone
312 555-1234
$
```

Or you can redirect an entire file of phone numbers to filter out the invalid ones:

```
$ cat phonelist
000-000-0000
123-456-7890
```

```
212-555-1234
(317) 555-1234
(202) 555-9876
33523
1234567890
234.123.4567
$ cat phonelist | ./isphone
212-555-1234
(317) 555-1234
(202) 555-9876
234.123.4567
$
```

Only the valid phone numbers that match the regular expression pattern appear.

### Parsing an e-mail address

These days, e-mail has become a crucial form of communication. Trying to validate e-mail addresses has become quite a challenge for script builders because of the myriad ways to create an e-mail address. This is the basic form of an e-mail address:

*username@hostname*

The *username* value can use any alphanumeric character, along with several special characters:

- Dot
- Dash
- Plus sign
- Underscore

These characters can appear in any combination in a valid e-mail userid. The *hostname* portion of the e-mail address consists of one or more domain names and a server name. The server and domain names must also follow strict naming rules, allowing only alphanumeric characters, along with the special characters:

- Dot
- Underscore

The server and domain names are each separated by a dot, with the server name specified first, any subdomain names specified next, and finally, the top-level domain name without a trailing dot.

At one time, the top-level domains were fairly limited, and regular expression pattern builders attempted to add them all in patterns for validation. Unfortunately, as the Internet grew, so did the possible top-level domains. This technique is no longer a viable solution.



Let's start building the regular expression pattern from the left side. We know that there can be multiple valid characters in the username. This should be fairly easy:

```
^([a-zA-Z0-9_-\.\+]+)@
```

This grouping specifies the allowable characters in the username and the plus sign to indicate that at least one character must be present. The next character obviously is the `@` symbol — no surprises there.

The hostname pattern uses the same technique to match the server name and the subdomain names:

```
([a-zA-Z0-9_-\.\+]+)
```

This pattern matches the text:

```
server
server.subdomain
server.subdomain.subdomain
```

There are special rules for the top-level domain. Top-level domains are only alphabetic characters, and they must be no fewer than two characters (used in country codes) and no more than five characters in length. The following is the regular expression pattern for the top-level domain:

```
\.([a-zA-Z]{2,5})$
```

Putting the entire pattern together results in the following:

```
^([a-zA-Z0-9_-\.\+]+)@([a-zA-Z0-9_-\.\+]+)\.([a-zA-Z]{2,5})$
```

This pattern filters out poorly formatted e-mail addresses from a data list. Now you can create your script to implement the regular expression:

```
$ echo "rich@here.now" | ./isemail
rich@here.now
$ echo "rich@here.now." | ./isemail
$
$ echo "rich@here.n" | ./isemail
$
$ echo "rich@here-now" | ./isemail
$
$ echo "rich.blum@here.now" | ./isemail
rich.blum@here.now
$ echo "rich_blum@here.now" | ./isemail
rich_blum@here.now
$ echo "rich/blum@here.now" | ./isemail
$
```

```
$ echo "rich#blum@here.now" | ./isemail
$
$ echo "rich*blum@here.now" | ./isemail
$
```

## Summary

---

If you manipulate data files in shell scripts, you need to become familiar with regular expressions. Regular expressions are implemented in Linux utilities, programming languages, and applications using regular expression engines. A host of different regular expression engines is available in the Linux world. The two most popular are the POSIX Basic Regular Expression (BRE) engine and the POSIX Extended Regular Expression (ERE) engine. The `sed` editor conforms mainly to the BRE engine, while the `gawk` program utilizes most features found in the ERE engine.

A regular expression defines a pattern template that's used to filter text in a data stream. The pattern consists of a combination of standard text characters and special characters. The special characters are used by the regular expression engine to match a series of one or more characters, similarly to how wildcard characters work in other applications.

By combining characters and special characters, you can define a pattern to match almost any type of data. You can then use the `sed` editor or `gawk` program to filter specific data from a larger data stream, or for validating data received from data entry applications.

The next chapter digs deeper into using the `sed` editor to perform advanced text manipulation. Lots of advanced features are available in the `sed` editor that make it useful for handling large data streams and filtering out just what you need.