



# Speaking UNIX, Part 9: Regular expressions

## Find and manipulate text

Level: Intermediate

Martin Streicher ([martin.streicher@gmail.com](mailto:martin.streicher@gmail.com)), Chief Technology Officer, McClatchy Interactive

17 Apr 2007

Virtually all non-trivial problems require you to filter good data from bad. Discover the many UNIX® command line utilities that use *regular expressions* to discern the relevant from the irrelevant.

Oddly enough, to this day, I can still repeat the Saturday morning classic "Conjunction Junction." Whether that's bad (way too much television) or good (perhaps a harbinger of my current career) is open for debate. Regardless, the little ditty conveyed foundational information in a happy little package.

I haven't come up with a "Conjunction Junction" equivalent for learning UNIX, but I'll try my hand at writing such a song in the coming months. In the meantime, and inspired by my happy memories, let's continue conquering the command line in the tradition of Schoolhouse Rock.

Class is now in session. Spit out your gum, take your seat, and find a No. 2 pencil. You, too, Mr. Spicoli.

### See Dick type. See Jane compute.

You can think of a UNIX command line as a sentence:

- An executable, such as *cat* or *ls*, is a **verb** -- an action.
- The output of a command is a **noun** -- data to be perused or used.
- A shell operator, such as *|* (pipe) or *>* (redirect standard output), is a **conjunction** -- a connector to link clauses.

For example, the command line: `ls -A | wc -l`, which counts the number of entries in the current directory (ignoring the special entries `.` and `..`), has two clauses. The first clause, `ls -A`, is a verb and enumerates the contents of the current directory; the second clause, `wc -l`, is another verb to count lines. The first produces output; the second consumes it, and a conjunction -- the pipe -- connects the two.

Many of the command line recipes shown previously in this series and others that you've likely cooked up have this sentence structure.

But without some grammatical gravy, the command line is as exciting as "See Dick type. See Jane compute." Sure, a primer sentence gets the job done, but it ain't *Hamlet*. (My apologies to Mrs. Rad and Mrs. Perlstein, the dynamic duo of senior-year English.) Solving more colorful problems requires *adjectives*.

Virtually all non-trivial problems require you to filter good data from bad. The number and kind of attributes might vary but, in some way, shape, or form, each solution (implicitly or explicitly) describes the information it seeks and processes that information, yielding more information in yet another form.

On the command line, the *regular expression* acts as an adjective -- a description or qualifier. When applied to output, the regular expression discerns between relevant data and craft.

### A little lesson in punctuation

Let's look at a sample problem.

The `grep` utility filters input line by line and looks for matches. In its simplest use, `grep` prints those lines that contain text that matches a pattern. `grep` can find fixed sequences of characters and can even ignore case with

the `-i` option.

Hence, given a file, `heroes.txt`, with these lines:

```
Catwoman
Batman
The Tick
Spider Man
Black Cat
Batgirl
Danger Girl
Wonder Woman
Luke Cage
The Punisher
Ant Man
Dead Girl
Aquaman
SCUD
Spider Woman
Blackbolt
Martian Manhunter
```

The command line:

```
grep -i man heroes.txt
```

produces:

```
Catwoman
Batman
Spider Man
Wonder Woman
Ant Man
Aquaman
Martian Manhunter
```

Here, `grep` scans each line in the `heroes.txt` file and looks for an *m*, followed by an *a*, and then followed by an *n*. Except for being contiguous, those letters can appear anywhere on the line, even embedded in a larger word. Catwoman, Batman, Spider Man, Wonder Woman, Ant Man, Aquaman, and Martian Manhunter each contain the string `man`, ignoring case (the `-i` option).

The `grep` utility has other built-in options to refine searches. For instance, the `-w` option restricts matches to whole words, so `grep -i -w man` would *exclude* Catwoman and Batman, for instance.

The tool also has a nice feature to exclude rather than include all matches found. Use the `-v` option to *omit* lines that match. For instance:

```
grep -v -i 'spider' heroes.txt
```

prints every line except those that contain the string `spider`:

```
Catwoman
Batman
The Tick
Black Cat
Batgirl
Danger Girl
Wonder Woman
Luke Cage
The Punisher
Ant Man
Dead Girl
Aquaman
SCUD
Blackbolt
Martian Manhunter
```

But, what if you want only those names that *begin* with "Bat"? Or words that begin "bat," "Bat," "cat," or "Cat?" Or perhaps you want to find how many comic avengers *end* with "man." In these cases, a simple string search, as performed in the previous three examples, doesn't suffice because the searches are insensitive to location.

## Location, location, location, and alternation

A regular expression *can* filter for a specific location, such as the start or end of a line and the beginning and end of a word. A regular expression, commonly abbreviated *regex*, can also describe alternates (which you might describe as "this" or "that"); fixed-, variable-, or indefinite-length repetition; ranges (for example, "any of the letters from a-m"); and classes, or kinds of characters ("printable characters" or "punctuation"), and other techniques.

Table 1 shows some common regular expression operators. You can string together the primitives in Table 1 (and other operators) and use them in combination to build (very) complex regular expressions.

**Table 1. Common regular expression operators**

Operator	Purpose
.	Match any single character.
^ (caret)	Match the empty string that occurs at the beginning of a line or string.
\$ (dollar sign)	Match the empty string that occurs at the end of a line.
A	Match an uppercase letter <i>A</i> .
a	Match a lowercase <i>a</i> .
\d	Match any single digit.
\D	Match any single non-digit character.
\w	Match any single alphanumeric character; a synonym is <code>[ :a\l num: ]</code> .
[A-E]	Match any of uppercase <i>A</i> , <i>B</i> , <i>C</i> , <i>D</i> , or <i>E</i> .
[^A-E]	Match any character <i>except</i> uppercase <i>A</i> , <i>B</i> , <i>C</i> , <i>D</i> , or <i>E</i> .
X?	Match no or one occurrence of the capital letter <i>X</i> .
X*	Match zero or more capital <i>X</i> s.
X+	Match one or more capital <i>X</i> s.
X{n}	Match exactly <i>n</i> capital <i>X</i> s.
X{n,m}	Match at least <i>n</i> and no more than <i>m</i> capital <i>X</i> s. If you omit <i>m</i> , the expression tries to match at least <i>n</i> <i>X</i> s.
(abc def)+	Match a sequence of at least one <i>abc</i> and <i>def</i> ; <i>abc</i> and <i>def</i> would match.

Here are a few examples of regular expressions using **grep** as the search tool. Many other UNIX tools, including interactive editors **vi** and Emacs, stream editors **sed** and **awk**, and all modern programming languages, also support regular expressions. Once you learn the (admittedly cryptic) syntax of regular expressions, you can transfer your expertise among tools, programming languages, and operating systems.

### Find names that begin with "Bat"

To find names that begin with "Bat," use:

```
grep -E '^Bat'
```

You use the `-E` option to specify the regular expression. The `^` (caret) character matches the beginning of a line or a string -- an imaginary character that appears before the first character of each line or string. The letters `B`, `a`, and `t` are literals and only match those specific characters. Hence, the command `grep -E '^Bat'` produces:

```
Batman
Batgirl
```

Because many regex operators are also used by the shell (some for different purposes and some for similar purposes), it's a good habit to surround each regex provided on the command line with single quotation marks to protect the regex operators from interpolation by the shell. For example, both `*` (asterisk) and `$` (dollar sign) are regex operators and also have special meaning to your shell.

### Find names that end with "man"

To find names that end with "man," you might use the regex `man$` to match the sequence `m`, `a`, and `n` followed immediately by the end of the line (or string) matched by the regex operator `$`.

### Find a blank line

Given the purpose of `^` and `$`, you can find a blank line using the regex `^$` -- essentially, a line that ends immediately after it begins.

### Alternation or the set operator

To find words that begin with "bat," "Bat," "cat," or "Cat," you can use one of two techniques. The first is *alternation*, which yields a match if *any* of the patterns in the alternation match. For example, the command:

```
grep -E '^(bat|Bat|cat|Cat)' heroes.txt
```

does the trick. The regex operator `|` (vertical bar) is the alternation, so `this|that` matches either the string `this` or the string `that`. Hence, `^(bat|Bat|cat|Cat)` specifies, "The beginning of a line followed immediately by one of `bat`, `Bat`, `cat`, or `Cat`." Of course, you could simplify the regex using `grep -i`, which ignores case, reducing the command to:

```
grep -i -E '^(bat|cat)' heroes.txt
```

The other approach to matching "bat," "Bat," "cat," or "Cat" uses the `[ ]` (brackets) *set* operator. If you place a list of characters in a set, any of those characters can match. (You can think of a *set* as shorthand for alternation of characters.)

For example, the command line:

```
grep -E '^[bcBC]at' heroes.txt
```

produces the same result as the command:

```
grep -E '^(bat|Bat|cat|Cat)' heroes.txt
```

You can simplify once again with `-i` to reduce the regex to `^[bc]at`.

Further, you can specify an inclusive range of characters in a set with the `-` (hyphen) operator. For instance, user names typically begin with a letter. To validate such a user name, say, in a Web form submitted to your server, you might use a regex such as `^[A-Za-z]`. This regex reads, "The start of a string followed immediately by any uppercase letter (`A-Z`) or any lowercase letter (`a-z`)." By the way, `[A-z]` is the same as `[A-Za-z]`.

You can also mix ranges and individual characters in a set. The regex `[A-MXYZ]` would match any of uppercase

A-M, X, Y, and Z.

And if you want the inverse of a set -- that is, any character except what's in the set -- use the special set `[^ ]` and include the range or characters to exclude. Here's an example of an inverse set. To find all superheroes with *at* in the name, excluding the Dark Knight, Batman, type:

```
grep -i -E '[^b]at' heroes.txt
```

The command produces:

```
Catwoman
Black Cat
```

Certain sets are required so frequently that shorthand notation has been developed to stand in for many. For instance, the set `[A-z0-9_]` is so common that it can be abbreviated `\w`. Likewise, the operator `\W` is a convenience for the set `[^A-z0-9_]`. You can also use the notation `[ :alnum: ]` instead of `\w` and `[ ^[:alnum:] ]` for `\W`.

By the way, `\w` (and synonym `[ :alnum: ]`) are specific to locale, while `[A-z0-9_]` is literally the letters A-z, the digits 0-9, and the underscore. If you're developing international applications, use the locale-specific forms to make your code portable among many locales.

## Repeat after me: Repetition, repetition, repetition

So far, you've seen literal, positional, and two kinds of alternation operators. With these alone, you can match most any pattern of a *predictable* length. Returning to user names, for example, you could ensure that every user name started with a letter and was followed by exactly seven letters or numbers through the regex:

```
[a-z][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9][a-z0-9]
```

But that's a little unwieldy. Moreover, it only matches user names of exactly eight characters. It won't match names between three and eight characters, which are also typical valid user names.

A regular expression can also include *repetition modifiers*. A repetition modifier can specify amounts such as none, one, or more; one or more; zero or one; five to ten; and exactly three. A repetition modifier must be combined with other patterns; the modifier has no meaning by itself.

As an example, the regex:

```
^[A-z][A-z0-9]{2,7}$
```

implements the user name filter desired earlier. A *user name* is a string beginning with a letter followed by at least two, but not more than seven letters or numbers followed by the end of the string.

The location anchors are essential here. Without the two positional operators, user names of arbitrary length would erroneously be accepted. Why? Consider the regex:

```
^[A-z][A-z0-9]{2,7}
```

It asks the question, "Does the string begin with a letter followed by two to seven letters?" But it makes no mention of the terminating condition. Thus, the string `samuelclemens` fits the criteria, but it is obviously too long for a valid user name. Similarly, omitting the beginning anchor, `^`, or both anchors would match strings that ended or contained something like `munster1313`, respectively. If your match must be a specific length, don't forget to include delimiters for the beginning and end of the desired pattern.

Here are a few other samples:

- You can use `{2,}` to find two or more repeats. The regex `^G[o]{2,}gle` matches `Google`, `Gooogle`, `Goooogle`, and so on.
- The repetition modifiers `?`, `+`, and `*` find no or one, one or more, and zero or more repeats, respectively. (You can think of `?` as a shorthand for `{0,1}`, for instance.)

The regex `boys?` matches `boy` or `boys`; the regex `Goo?gle` matches `Gogle` or `Google`.

The regex `Goo+gle` matches `Google`, `Gooogle`, `Goooogle`, and so on.

The construct `Goo*gle` matches `Gogle`, `Google`, `Gooogle`, and so on.

- You can apply repetition modifiers to individual literals, as shown immediately above, as well as to other, more complex combinations. Use `(` and `)` parentheses (just as you do in mathematics) to apply a modifier to a subexpression. Here's an example: Given text file `test.txt`:

```
The rain in Spain falls mainly
on the the plain.

It was the best of of times;
it was the worst of times.
```

the command `grep -i -E '(\b(of|the)\W+){2,}' test.txt` produces:

```
on the the plain.
It was the best of of times;
```

- The regex operator `\b` matches a *word boundary* or `(\W\w|\w\W)`. The regex reads, "A sequence of whole words 'the' or 'of' followed by a non-word character." You might be asking why the `\W+` is necessary: `\b` is the empty string at the beginning or end of a word. You have to include the character or characters between the words, otherwise the regex fails to find a match.

## Capture what needs your attention

Finding text is a common problem but, more often than not, you want to extract the text after it's found. In other words, you want to keep the needle and discard the haystack.

A regular expression extracts information through *capture*. If you want to isolate the text you want from craft, surround the pattern you want with parentheses. Indeed, you already used parentheses to collect terms; by default, parentheses capture automatically.

To see capture, let's switch to Perl. (The `grep` utility does not support capture, because its purpose is to print lines containing a pattern.)

The command:

```
perl -n -e '/^The\s+(.*)$/ && print "$1\n"' heroes.txt
```

prints:

```
Tick
Punisher
```

The command `perl -e` lets you run a Perl program right from the command line. The `perl -n` command runs the program once on every line of the input file. The regex portion of the command, the text between the slashes (`/`), says, "Match the beginning of string, then literals 'T', 'h', 'e,' followed by one or more whitespace character or characters, `\s+`, and then capture every character to the end of the string.

Perl captures are placed in special Perl variables beginning with `$1`. The rest of the Perl program prints what was captured.

Each nested set of parentheses, counting from the left, incrementing at each left parenthesis is placed in the next special, numerical variable. For example:

```
perl -n -e '/^(\w)+-(\w+)$/ && print "$1 $2"'
```

yields:

```
Spider Man  
Ant Man  
Spider Woman
```

Capturing text of interest just scratches the surface. When you can pinpoint material, you can surgically replace it with other material. Editors such as `vi` and Emacs combine pattern matching and substitution to find and replace text in one fell swoop. You can also alter text from the command line using patterns, `replace`, and `sed`.

## A rich topic

Regular expressions are extremely powerful; the number and kind of operators and techniques you can command are enormous. There's so much information and practical knowledge that it's impossible to present but a fraction here.

Luckily, three excellent sources of regular expression theory and practice are available:

- If you have Perl on your system, consult the Perl Regular Expression man page (type `perldoc perlre`). It provides an excellent introduction to regex and has many useful examples. Many programming languages have adopted Perl Compatible Regular Expressions (PCRE), so what you read in this man page translates directly to the PHP, Python, Java™, Ruby programming languages, as well as many other modern tools.
- Jeffrey Friedl's book, *Regular Expressions* (third edition), is considered the bible of regex use. Meticulous, precise, clear, and practical, the book explains how matches work, all the regex operators, greediness (restricting how many characters `+` and `*` match), and much more. Moreover, Friedl's book includes some truly mind-blowing regular expressions to properly match fully qualified e-mail addresses and other Request for Comments- (RFC) specified strings.
- Nathan Good's book, *Regular Expression Recipes*, presents helpful solutions to many common data processing and filtering problems. Need to extract a zip code, phone number, or quoted string? Try Nathan's solutions.

At the command line, you'll find many ways to use regular expressions. Virtually every command that processes text supports regular expressions of one form or another. Most shell command syntax also expands regular expressions to match file names, although the operators might function differently, greatly, or slightly.

For example, type `ls [a-c]` to find the files named *a*, *b*, and *c*. Typing `ls [a-c]*` finds all file names that begin with *a*, *b*, or *c*. Here, the `*` does not modify the `[a-c]` as `grep`'s interpreter would; in the shell, `*` is interpreted as `.*`. The `?` operator works in the shell, too, but is interpreted as `.`, or match any single character.

Check the documentation for your favorite utility or shell to determine which regex operators are supported and how, if at all, the operators are unique.

## School's out!

This lesson was longer than usual. But now you know the ABCs of regular expressions. Go out and express yourself.

While you enjoy recess, I'll start working on the soon-to-be pop classic "99 Lines About 99 Commands."

## Resources

### Learn

- [Speaking UNIX](#): Check out other parts in this series.
- [AIX and UNIX](#): The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- [New to AIX and UNIX?](#): Visit the New to AIX and UNIX page to learn more about AIX and UNIX.
- [AIX 5L™ Wiki](#): A collaborative environment for technical information related to AIX.
- Check out other articles and tutorials written by Martin Striecher:
  - [Across developerWorks and IBM](#)
- Search the AIX and UNIX library by topic:
  - [System administration](#)
  - [Application development](#)
  - [Performance](#)
  - [Porting](#)
  - [Security](#)
  - [Tips](#)
  - [Tools and utilities](#)
  - [Java™ technology](#)
  - [Linux](#)
  - [Open source](#)
- [Safari bookstore](#): Visit this e-reference library to find specific technical resources.

### Get products and technologies

- [IBM trial software](#): Build your next development project with software for download directly from developerWorks.

### Discuss

- Participate in the [developerWorks blogs](#) and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - [AIX 5L -- technical forum](#)
  - [AIX for Developers Forum](#)
  - [Cluster Systems Management](#)
  - [IBM Support Assistant](#)
  - [Performance Tools -- technical](#)
  - [Virtualization -- technical](#)
  - [More AIX and UNIX forums](#)
- [zsh](#): Collaborate, discuss, and share your expertise of zsh on the zsh wiki.

### About the author





Martin Streicher is the Chief Technology Officer of McClatchy Interactive and the Editor-in-Chief of *Linux Magazine*. Martin holds a Masters of Science degree in computer science from Purdue University and has been programming UNIX-like systems since 1986. You can reach Martin at [martin.streicher@gmail.com](mailto:martin.streicher@gmail.com).

### Share this....



[Digg this story](#)



[del.icio.us](#)



[Slashdot it!](#)

UNIX is a registered trademark of The Open Group in the United States and other countries. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.