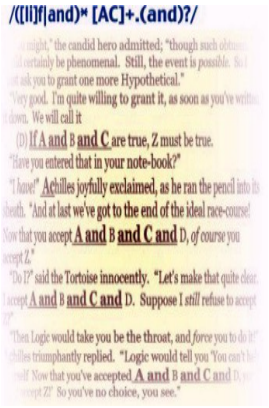
Tutorial: Using regular expressions

Section 1. Introduction to the tutorial



Who is this tutorial for?

This tutorial is aimed at programmers who work with tools that use regular expressions, and who would like to become more comfortable with the intricacies of regular expressions. Even programmers who have used regular expressions in the past, but have forgotten some of the details, can benefit from this tutorial as a refresher.

After completing this tutorial, you will not yet be an expert in using regular expressions to best advantage. But this tutorial combined with **lots** of practice with varying cases is about all you need to be an expert. The *concepts* of regular expressions are extremely simple and powerful -- it is their application that takes some work.

Just what is a regular expression, anyway?

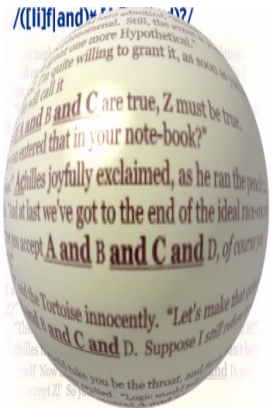
Take the tutorial to get the long answer. The short answer is that a regular expression is a compact way of describing complex patterns in texts. You can use them to search for patterns and, once found, to modify the patterns in complex ways. You can also use them to launch programmatic actions that depend on patterns.

A tongue-in-cheek comment by programmers is worth thinking about: *"Sometimes you have a programming problem and it seems like the best solution is to use regular expressions; now you have two problems."* Regular expressions are amazingly powerful and deeply expressive. That is the very reason writing them is just as error-prone as writing any other complex programming code. It is always better to solve a genuinely simple problem in a simple way; when you go beyond simple, think about regular expressions.

What tools use regular expressions?

Many tools incorporate regular expressions as part of their functionality. UNIX-oriented command line tools like grep, sed, and awk are mostly wrapper for regular-expression processing. Many text editors allow search and/or replacement based on regular expressions. Many programming languages, especially scripting languages such as Perl, Python, and TCL, build regular expressions into the heart of the language. Even most command-line shells, such as Bash or the Windows-console, allow restricted regular expressions as part of their command syntax.

There are a few variations in regular-expression syntax between different tools that use them. Some tools add enhanced capabilities that are not available everywhere. In general, for the simplest cases, this tutorial will use examples based around grep or sed. For a few more exotic capabilities, Perl or Python examples will be chosen. For the most part, examples will work anywhere; but check the documentation on your own tool for syntax variations and capabilities.



Note on presentation

For purposes of presenting examples in this tutorial, regular expressions described will be surrounded by forward slashes. This style of delimiting regular expressions is used by sed, awk, Perl, and other tools. For instance, an example might mention:

```
/[A-Z]+(abc|xyz)*/
```

Read ahead to understand this example, for now just understand that the actual regular expression is everything **between** the slashes.

Many examples will be accompanied by an illustration that shows a regular expression, and text that is highlighted for every match on that expression.

Tutorial navigation

Navigating through the tutorial is easy:

- Select Next and Previous to move forward and backward through the tutorial.
- When you're finished with a section, select the Main menu for the next section. Within a section, use the Section menu.
- If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Contact

David Mertz is a writer, a programmer, and a teacher who always endeavors to improve his communication to readers (and tutorial takers). He welcomes any comments; please direct them to *mertz@gnosis.cx*.

Section 2. Basic pattern matching in text

```
/a/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.

/Mary/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.
```

Character literals

The very simplest pattern matched by a regular expression is a literal character or a sequence of literal characters. Anything in the target text that consists of exactly those characters in exactly the order listed will match. A lowercase character is not identical to its uppercase version, and vice versa. A space in a regular expression, by the way, matches a literal space in the target (this is unlike most programming languages or command-line tools, where spaces separate keywords).

```
/.*/

Special characters must be escaped.*

/\.\*/
Special characters must be escaped.*
```

"Escaped" characters literals

A number of characters have special meanings to regular expressions. A symbol with a special meaning can be matched, but to do so you must prefix it with the backslash character (this includes the backslash character itself: to match one backslash in the target, your regular expression should include "\\").

```
/^Mary/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.

/Mary$/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.
```

Positional special characters

Two special characters are used in almost all regular expression tools to mark the beginning and end of a line: caret (^) and dollar-sign ($). To match a caret or dollar-sign as a literal character, you must escape it (that is, precede it with a backslash "\").

An interesting thing about the caret and dollar-sign is that they match **zero-width** patterns. That is, the length of the string matched by a caret or dollar-sign by itself is zero (but the rest of the regular expression can still depend on the zero-width match). Many regular expression tools provide another zero-width pattern for word-boundary (\b). Words might be divided by whitespace like spaces, tabs, newlines, or other characters like nulls; the word-boundary pattern matches the actual point where a word starts or ends, not the particular whitespace characters.

```
/.a/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.
```

The "wildcard" character

In regular expressions, a period can stand for any character. Normally, the newline character is not included, but most tools have optional switches to force inclusion of the newline character also. Using a period in a pattern is a way of requiring that "something" occurs here, without having to decide what.

Users who are familiar with DOS command-line wildcards will know the question-mark as filling the role of "some character" in command masks. But in regular expressions, the question-mark has a different meaning, and the period is used as a wildcard.

```
/(Mary)( )(had)/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.
```

Grouping regular expressions

A regular expression can have literal characters in it, and also zero-width positional patterns. Each literal character or positional pattern is an **atom** in a regular expression. You may also group several atoms together into a small regular expression that is part of a larger regular expression. One might be inclined to call such a grouping a "molecule," but normally it is also called an atom.

In older UNIX-oriented tools like grep, subexpressions must be grouped with escaped parentheses, as in `/\(Mary\)/`. In Perl and most more recent tools (including egrep), grouping is done with bare parentheses, but matching a literal parenthesis requires escaping it in the pattern (the example follows the Perl style).

```
/[a-z]a/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.
```

Character classes

Rather than name only a single character, you can include a pattern in a regular expression that matches any of a set of characters.

A set of characters can be given as a simple list inside square brackets; for example, `/[aeiou]/` will match any single lowercase vowel. For letter or number ranges you may also use only the first and last letter of a range, with a dash in the middle; for example, `/[A-Ma-m]/` will match any lowercase or uppercase in the first half of the alphabet.

Many regular expression tools also provide escape-style shortcuts to the most commonly used character class, such as `\w` for a whitespace character and `\d` for a digit. You could always define these character classes with square brackets, but the shortcuts can make regular expressions more compact and readable.

```
/[^a-z]a/

Mary had a little lamb.
And everywhere that Mary
went, the lamb was sure
to go.
```

Complement operator

The caret symbol can actually have two different meanings in regular expressions. Most of the time, it means to match the zero-length pattern for line beginnings. But if it is used at the beginning of a character class, it reverses the meaning of the character class. Everything **not** included in the listed character set is matched.

```
/cat|dog|bird/

The pet store sold cats, dogs, and
s.

/=xxx|yyy=/

=xxx xxx= # =yyy yyy= # =xxx= # =yyy=

/(=)(xxx)|(yyy)(=)/

=xxx xxx= # =yyy yyy= # =xxx= # =yyy=

/=(xxx|yyy)=/

=xxx xxx= # =yyy yyy= # =xxx= # =yyy=
```

Alternation of patterns

Using character classes is a way of indicating that either one thing or another thing can occur in a particular spot. But what if you want to specify that either of two whole subexpressions occurs in a position in the regular expression? For that, you use the alternation operator, the vertical bar ("|"). This is the symbol that is also used to indicate a pipe in UNIX/DOS shells, and is sometimes called the pipe character.

The pipe character in a regular expression indicates an alternation between **everything** in the group enclosing it. Even if there are several groups to the left and right of a pipe character, the alternation greedily asks for everything on both sides. To select the scope of the alternation, you must define a group that encompasses the patterns that may match. The example illustrates this.

```
/@(=+=)*@/

Match with zero in the middle: @@
Subexpression occurs, but...: @=+=ABC@
Many occurrences: @=+==+==+==+==+=@
Repeat entire pattern: @=+==+=+==+=@
```

The basic abstract quantifier

One of the most powerful and common things you can do with regular expressions is specify how many times an atom occurs in a complete regular expression. Sometimes you want to specify something about the occurrence of a single character, but very often you are interested in specifying the occurrence of a character class or a grouped subexpression.

There is only one quantifier included with "basic" regular expression syntax, the asterisk ("*"); this has the meaning "some or none" or "zero or more." If you want to specify that any number of an atom may occur as part of a pattern, follow the atom by an asterisk.

Without quantifiers, grouping expressions doesn't really serve much purpose, but once we can add a quantifier to a subexpression we can say something about the occurrence of the subexpression as a whole. Take a look at the example.

Section 3. Intermediate pattern matching in text

```
/A+B*C?D

AAAD
ABBBBCD
BBBCD
ABCCD
AAABBBC
```

More abstract quantifiers

In a way, the lack of any quantifier symbol after an atom quantifies the atom anyway: it says the atom occurs **exactly once**. Extended regular expressions (which most tools support) add a few other useful numbers to "once exactly" and "zero or more times." The plus-sign ("+") means "one or more times" and the question-mark ("?") means "zero or one times." These quantifiers are by far the most common enumerations you wind up naming.

If you think about it, you can see that the extended regular expressions do not actually let you "say" anything the basic ones do not. They just let you say it in a shorter and more readable way. For example, "(ABC)+" is equivalent to (ABC)(ABC)*"; and "X(ABC)?Y" is equivalent to XABCY|XY". If the atoms being quantified are themselves complicated grouped subexpressions, the question-mark and plus-sign can make things a lot shorter.

```
/a{5} b{,6} c{4,8}/

aaaaa bbbbb ccccc
aaa bbb ccc
aaaaa bbbbbbbbbbbbb ccccc

/a+ b{3,} c?/

aaaaa bbbbb ccccc
aaa bbb ccc
aaaaa bbbbbbbbbbbbb ccccc

/a{5} b{6,} c{4,8}/

aaaaa bbbbb ccccc
aaa bbb ccc
aaaaa bbbbbbbbbbbbb ccccc
```

Numeric quantifiers

Using extended regular expressions, you can specify arbitrary pattern occurrence counts using a more verbose syntax than the question-mark, plus-sign, and asterisk quantifiers. The curly-braces ("{" and "}") can surround a precise count of how many occurrences you are looking for.

The most general form of the curly-brace quantification uses two range arguments (the first must be no larger than the second, and both must be non-negative integers). The occurrence count is specified this way to fall between the minimum and maximum indicated (inclusive). As shorthand, either argument may be left empty: if so, the minimum/maximum is specified as zero/infinity, respectively. If only one argument is used (with no comma in there), **exactly** that many occurrences are matched.

```
/(abc|xyz) \1/

jkl abc xyz
jkl xyz abc
jkl abc abc
jkl xyz xyz

/(abc|xyz) (abc|xyz)/

jkl abc xyz
jkl xyz abc
jkl abc abc
jkl xyz xyz
```

Backreferences

One powerful option in creating search patterns is specifying that a subexpression that was matched earlier in a regular expression is matched again later in the expression. We do this using **backreferences**. Backreferences are named by the numbers 1 through 9, preceded by the backslash/escape character when used in this manner. These backreferences refer to each successive group in the match pattern, as in `/(one)(two)(three)/\1\2\3/`. Each numbered backreference refers to the group that, in this example, has the word corresponding to the number.

It is important to note something the example illustrates. What gets matched by a backreference is the same literal string matched the first time, even if the pattern that matched the string could have matched other strings. Simply repeating the same grouped subexpression later in the regular expression does not match the same targets as using a backreference (but you have to decide what you actually want to match in either case).

Backreferences refer back to whatever occurred in the previous grouped expressions, in the order those grouped expressions occurred. Because of the naming convention (\1-\9), many tools limit you to nine backreferences. Some tools allow actual naming of backreferences and/or saving them to program variables. Section 4 touches on these topics.

```
/th.*s/

-- Match the words that start
-- with 'th' and end with 's'.
this
thus
thistle
this line matches too much
```

Don't match more than you want to

Quantifiers in regular expressions are **greedy**. That is, they match as much as they possibly can.

Probably the easiest mistake to make in composing regular expressions is to match **too much**. When you use a quantifier, you want it to match everything (of the right sort) up to the point where you want to finish your match. But when using the "*", "+", or numeric quantifiers, it is easy to forget that the last bit you are looking for might occur later in a line than the one you are interested in.

```
/th.*s/

-- Match the words that start
-- with 'th' and end with 's'.

/th[^s]*./

-- Match the words that start
-- with 'th' and end with 's'.
this
thus
thistle
this line matches too much
```

Tricks for restraining matches

If you find that your regular expressions are matching too much, a useful procedure is to reformulate the problem in your mind. Rather than thinking "what am I trying to match later in the expression?" ask yourself "what do I need to **avoid** matching in the next part?". Often this leads to more parsimonious pattern matches. Often the way to avoid a pattern is to use the complement operator and a character class. Look at the example, and think about how it works.

The trick here is that there are two different ways of formulating **almost** the same sequence. You can either think you want to keep matching **until** you get to XYZ, or you can think you want to keep matching **unless** you get to XYZ. These are subtly different.

For people who have thought about basic probability, the same pattern occurs. The chance of rolling a 6 on a die in one roll is 1/6. What is the chance of rolling a 6 in six rolls? A naive calculation puts the odds at 1/6+1/6+1/6+1/6+1/6+1/6, or 100%. This is wrong, of course (after all, the chance after twelve rolls isn't 200%). The correct calculation is "how do I **avoid** rolling a 6 for six rolls?" -- in other words, 5/6*5/6*5/6*5/6*5/6*5/6, or about 33%. The chance of **getting** a 6 is the same chance as not **avoiding** it (or about 66%). In fact, if you imagine transcribing a series of dice rolls, you could apply a regular expression to the written record, and similar thinking applies.

Comments on modification tools

Not all tools that use regular expressions allow you to modify target strings. Some simply locate the matched pattern; the mostly widely used regular expression tool is probably grep, which is a tool for searching only. Text editors, for example, may or may not allow replacement in their regular expression search facility. As always, consult the documentation on your individual tool.

Of the tools that allow you to modify target text, there are a few differences to keep in mind. The way you actually specify replacements will vary between tools: a text editor might have a dialog box; command-line tools will use delimiters between match and replacement, programming languages will typically call functions with arguments for match and replacement patterns.

Another important difference to keep in mind is what is getting modified. UNIX-oriented command-line tools typically utilize pipes and STDOUT for changes to buffers, rather than modify files in-place. Using a sed command, for example, will write the modifications to the console, but will not change the original target file. Text editors or programming languages are more likely to actually modify a file in-place.

A note on modification examples

For purposes of this tutorial, examples will continue to use the sed style slash delimiters. Specifically, the examples will indicate the substitution command and the global modifier, as with "`s/this/that/g`". This expression means: "Replace the string 'this' with the string 'that' everywhere in the target text.

Examples will consist of the modification command, an input line, and an output line. The output line will have any changes emphasized. Also, each input/output line will be preceded by a less-than or greater-than symbol to help distinguish them (the order will be as described also), which is suggestive of redirection symbols in UNIX shells.

A literal-string modification example

Let's take a look at a few modification examples that build on what we have already covered.

```
s/cat/dog/g

< wild dogs, bobcats, lions, and other wild cats
> wild dogs, bob**dog**s, lions, and other wild **dog**s
```

This one simply substitutes some literal text for some other literal text. The search-and-replace capability of many tools can do this much, even without using regular expressions.

A pattern-match modification example

```
s/cat|dog/snake/g

< wild dogs, bobcats, lions, and other wild cats
> wild **snake**s, bob**snake**s, lions, and other wild **snake**s

s/[a-z]+i[a-z]*/nice/g

< wild dogs, bobcats, lions, and other wild cats
> **nice** dogs, bobcats, **nice**, and other **nice** cats
```

Most of the time, if you are using regular expressions to modify a target text, you will want to match more general patterns than just literal strings. Whatever is matched is what gets replaced (even if it is several different strings in the target).

```
s/([A-Z])([0-9]{2,4}) /\2:\1 /g

< A37 B4 C107 D54112 E1103 XXX
> 37:A B4 107:C D54112 1103:E XXX
```

Modification using backreferences

It is nice to be able to insert a fixed string everywhere a pattern occurs in a target text. But frankly, doing that is not very context sensitive. A lot of times, we do not want just to insert fixed strings, but rather to insert something that bears much more relation to the matched patterns. Fortunately, backreferences come to our rescue here. You can use backreferences in the pattern-matches themselves, but it is even more useful to be able to use them in replacement patterns. By using replacement backreferences, you can pick and choose from the matched patterns to use just the parts you are interested in.

To aid readability, subexpressions will be grouped with bare parentheses (as with Perl), rather than with escaped parentheses (as with sed).

Another warning on mismatching

This tutorial has already warned about the danger of matching too much with your regular expression patterns. But the danger is so much more serious when you do modifications, that it is worth repeating. If you replace a pattern that matches a larger string than you thought of when you composed the pattern, you have potentially deleted some important data from your target.

It is always a good idea to try out your regular expressions on diverse target data that is representative of your production usage. Make sure you are matching what you think you are matching. A stray quantifier or wildcard can make a surprisingly wide variety of texts match what you thought was a specific pattern. And sometimes you just have to stare at your pattern for a while, or find another set of eyes, to figure out what is really going on even after you see what matches. Familiarity might breed contempt, but it also instills competence.

Section 4. Advanced regular expression extensions

About advanced features

Some very useful enhancements are included in some regular expression tools. These enhancements often make the composition and maintenance of regular expression considerably easier. But check with your own tool to see what is supported.

The programming language Perl is probably the most sophisticated tool for regular-expression processing, which explains much of its popularity. The examples illustrated will use Perl-ish code to explain concepts. Other programming languages, especially other scripting languages such as Python, have a similar range of enhancements. But for purposes of illustration, Perl's syntax most closely mirrors the regular expression tools it builds on, such as ed, ex, grep, sed, and awk.

```
/th.*s/

-- Match the words that start
-- with 'th' and end with 's'.
this line matches just right
this # thus # thistle

/th.*?s/

-- Match the words that start
-- with 'th' and end with 's'.
this # thus # thistle
this line matches just right

/th.*?s /

-- Match the words that start
-- with 'th' and end with 's'.
-- (FINALLY!)S
this # thus # thistle
this line matches just right
```

Non-greedy quantifiers

Earlier in the tutorial, the problems of matching too much were discussed, and some workarounds were suggested. Some regular expression tools make this easier by providing optional **non-greedy** quantifiers. These quantifier grab as little as possible while still matching whatever comes next in the pattern (instead of **as much** as possible).

Non-greedy quantifiers have the same syntax as regular greedy ones, except with the quantifier followed by a question-mark. For example, a non-greedy pattern might look like: `/A[A-Z]*?B/`". In English, this means "match an A, followed by only as many capital letters as are needed to find a B."

One little thing to look out for is the fact that the pattern "`/[A-Z]*?./`" will always match **zero** capital letters. If you use non-greedy quantifiers, watch out for matching **too little**, which is a symmetric danger.

```
/M.*[ise] /

MAINE # Massachusetts # Colorado #
mississippi # Missouri # Minnesota #

/M.*[ise] /i

MAINE # Massachusetts # Colorado #
mississippi # Missouri # Minnesota #

/M.*[ise] /gis

MAINE # Massachusetts # Colorado #
mississippi # Missouri # Minnesota #
```

Pattern-match modifiers

We already saw one pattern-match modifier in the modification examples: the **g**lobal modifier. In fact, in many regular expression tools, we should have been using the "g" modifier for all our pattern matches. Without the "g", many tools will match only the first occurrence of a pattern on a line in the target. So this is a useful modifier (but not one you necessarily want to use always). Let us look at some others.

As a little mnemonic, it is nice to remember the word "gismo" (it even seems somehow appropriate). The most frequent modifiers are:

- g - Match globally
- i - Case-insensitive match
- s - Treat string as single line
- m - Treat string as multiple lines
- o - Only compile pattern once

The **o** option is an implementation optimization, and not really a regular expression issue (but it helps the mnemonic). The **s**ingle-line option allows the wildcard to match a newline character (it won't otherwise). The ultiple-line option causes "^" and "$" to match the begin and end of each line in the target, not just the begin/end of the target as a whole (with sed or grep this is the default). The **i**nsensitive option ignores differences between case of letters.

```
s/([A-Z])(?:-[a-z]{3}-)([0-9]*)/\1\2/g

< A-xyz-37 # B:abcd:142 # C-wxy-66
> A37 # B:abcd:42 # C66
```

Changing backreference behavior

Backreferencing in replacement patterns is very powerful; but it is also easy to use more than nine groups in a complex regular expression. Quite apart from using up the available backreference names, it is often more legible to refer to the parts of a replacement pattern in sequential order. To handle this issue, some regular expression tools allow "grouping without backreferencing."

A group that should not also be treated as a backreference has a question-mark colon at the beginning of the group, as in "`(?:pattern)`." In fact, you can use this syntax even when your backreferences are in the search pattern itself.

Naming backreferences

```
import re
txt = "A-xyz-37 # B:abcd:142 # C-wxy-66 # D-qrs-93"
new=re.sub("(?P<pre>[A-Z])(-[a-z]{3}-)(?P<id>[0-9]*)",
            "\g<pre>\g<id>", txt)
print new

A37 # B:abcd:42 # C66 # D93
```

The language Python offers a particularly handy syntax for really complex pattern backreferences. Rather than just play with the numbering of matched groups, you can give them a name.

The syntax of using regular expressions in Python is a standard programming language function/method style of call, rather than Perl- or sed-style slash delimiters. Check your own tool to see if it supports this facility.

```
s/([A-Z]-)(?=[a-z]{3})([a-z0-9]* )/\2\1/g

< A-xyz37 # B-ab6142 # C-Wxy66 # D-qrs93
> xyz37A- # B-ab6142 # C-Wxy66 # qrs93D-

s/([A-Z]-)(!=[a-z]{3})([a-z0-9]* )/\2\1/g

< A-xyz37 # B-ab6142 # C-Wxy66 # D-qrs93
> A-xyz37 # ab6142B- # Wxy66C- # D-qrs93
```

Lookahead assertions

Another trick of advanced regular expression tools is "lookahead assertions." These are similar to regular grouped subexpression, except they do not actually grab what they match. There are two advantages to using lookahead assertions. On the one hand, a lookahead assertion can function in a similar way to a group that is not backreferenced; that is, you can match something without counting it in backreferences. More significantly, however, a lookahead assertion can specify that the next chunk of a pattern has a certain form, but let a different subexpression actually grab it (usually for purposes of backreferencing that other subexpression).

There are two kinds of lookahead assertions: positive and negative. As you would expect, a positive assertion specifies that something **does** come next, and a negative one specifies that something **does not** come next. Emphasizing their connection with non-backreferenced groups, the syntax for lookahead assertions is similar: `(?=pattern)` for positive assertions, and `(?!pattern)` for negative assertions.

Making regular expressions more readable

```
/               # identify URLs within a text file
          [^="] # do not match URLs in IMG tags like:
                # <img src="http://this.com/pic.png">
http|ftp|gopher # make sure we find a resource type
          :\/\/ # ...followed by colon-slash-slash
      [^ \n\r]+ # not space, newline, or tab in URL
    (?=[\s\.,]) # assert next: whitespace/period/comma
/

The URL for my site is: http://mysite.com/mydoc.html.  You
might also enjoy ftp://yoursite.com/index.html for a good
place to download files.
```

In the later examples we have started to see just how complicated regular expressions can get. These examples are not the half of it. It is possible to do some almost absurdly difficult-to-understand things with regular expression (but things that are nonetheless useful).

There are two basic facilities that some of the more advanced regular expression tools use in clarifying expressions. One is allowing regular expressions to continue over multiple lines (by ignoring whitespace like trailing spaces and newlines). The second is allowing comments within regular expressions. Some tools allow you to do one or another of these things alone, but when it gets complicated, do both!

The example given uses Perl's e**x**tend modifier to enable commented multi-line regular expressions. Consult the documentation for your own tool for details on how to compose these.

Section 5. Summary

Resources

You have seen the basics (and a bit of some advanced topics) of regular expressions. The best thing to do next is to start using them in real-life problems. The first thing to look at is the documentation that accompanies the particular tool you use. Beyond that, a number of books have good explanations of regular expressions, often as implemented by specific tools. I have benefited from these:

- *Mastering Regular Expressions*, Jeffrey E. F. Friedl, O'Reilly, Cambridge, MA; 1997
- *sed & awk*, Dale Dougherty and Arnold Robbins, O'Reilly, Cambridge, MA; 1997
- *Programming Perl*, Larry Wall, Tom Christiansen and Randal L. Schwartz, O'Reilly, Cambridge, MA; 1996
- *TCL/TK in a Nutshell*, Paul Raines and Jeff Tranter, O'Reilly, Cambridge, MA; 1999
- *Python Pocket Reference*, Mark Lutz, O'Reilly, Cambridge, MA; 1998
- *A Practical Guide to Linux*, Mark G. Sobell, Addison Wesley, Reading, MA; 1997

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

For questions about the content of this tutorial, contact the author, David Mertz, at *mertz@gnosis.cx*.

Colophon

This tutorial was written entirely in XML, using the developerWorks tutorial tag set. The tutorial is converted into a number of HTML pages, a zip file, JPEG heading graphics, and a PDF file by a Java program and a set of XSLT stylesheets.