The 30 Minute Regex Tutorial

Jim Hollenhorst, 21 Nov 2005


4.96 (697 votes)

Learn how to use regular expressions in 30 minutes with Expresso.

Expresso 2.1C - 328 Kb

Learning .NET Regular Expressions with Expresso

Did you ever wonder what Regular Expressions are all about and want to gain a basic understanding quickly? My goal is to get you up and running with a basic understanding of regular expressions within 30 minutes. The reality is that regular expressions aren't as complex as they look. The best way to learn is to start writing and experimenting. After your first half hour, you should know a few of the basic constructs and be able to design and use regular expressions in your programs or web pages. For those of you who get hooked, there are many excellent resources available to further your education.

What the Heck is a Regular Expression Anyway?

I'm sure you are familiar with the use of "wildcard" characters for pattern matching. For example, if you want to find all the Microsoft Word files in a Windows directory, you search for "`*.doc`", knowing that the asterisk is interpreted as a wildcard that can match any sequence of characters. Regular expressions are just an elaborate extension of this capability.

In writing programs or web pages that manipulate text, it is frequently necessary to locate strings that match complex patterns. Regular expressions were invented to describe such patterns. Thus, a regular expression is just a shorthand code for a pattern. For example, the pattern "\w+" is a concise way to say "match any non-null strings of alphanumeric characters". The .NET framework provides a powerful class library that makes it easy to include regular expressions in your applications. With this library, you can readily search and replace text, decode complex headers, parse languages, or validate text.

A good way to learn the arcane syntax of regular expressions is by starting with examples and then experimenting with your own creations. This tutorial introduces the basics of regular expressions, giving many examples that are included in an Expresso library file. Expresso can be used to try out the examples and to experiment with your own regular expressions.

Let's get started!

Some Simple Examples

Searching for Elvis

Suppose you spend all your free time scanning documents looking for evidence that Elvis is still alive. You could search with the following regular expression:

1. `elvis` *Find elvis*

This is a perfectly valid regular expression that searches for an exact sequence of characters. In .NET, you can easily set options to ignore the case of characters, so this expression will match "Elvis", "ELVIS", or "eLvIs". Unfortunately, it will also match the last five letters of the word "pelvis". We can improve the expression as follows:

2. `\belvis\b` *Find elvis as a whole word*

Now things are getting a little more interesting. The "`\b`" is a special code that means, "match the position at the beginning or end of any word". This expression will only match complete words spelled "elvis" with any combination of lower case or capital letters.

Suppose you want to find all lines in which the word "elvis" is followed by the word "alive." The period or dot "`.`" is a special code that matches any character other than a newline. The asterisk "`*`" means repeat the previous term as many times as necessary to guarantee a match. Thus, "`.*`" means "match any number of characters other than newline". It is now a simple matter to build an expression that means "search for the word 'elvis' followed on the same line by the word 'alive'."

3. `\belvis\b.*\balive\b` *Find text with "elvis" followed by "alive"*

With just a few special characters we are beginning to build powerful regular expressions and they are already becoming hard for we humans to read.

Let's try another example.

Determining the Validity of Phone Numbers

Suppose your web page collects a customer's seven-digit phone number and you want to verify that the phone number is in the correct format, "xxx-xxxx", where each "x" is a digit. The following expression will search through text looking for such a string:

4. `\b\d\d\d-\d\d\d\d` *Find seven-digit phone number*

Each "`\d`" means "match any single digit". The "`-`" has no special meaning and is interpreted literally, matching a hyphen. To avoid the annoying repetition, we can use a shorthand notation that means the same thing:

5. `\b\d{3}-\d{4}` *Find seven-digit phone number a better way*

The "`{3}`" following the "`\d`" means "repeat the preceding character three times".

Let's learn how to test this expression.

Expresso

If you don't find regular expressions hard to read you are probably an idiot savant or a visitor from another planet. The syntax can be imposing for anyone, including those who use regular expressions frequently. This makes errors common and creates a need for a simple tool for building and testing expressions. Many such tools exist, but I'm partial to my own, Expresso, originally launched on the CodeProject. Version 2.0 is shown here. For later versions, check the Ultrapico website.

To get started, install Expresso and select the Tutorial from the Windows Program menu. Each example can be selected using the tab labeled "Expression Library".

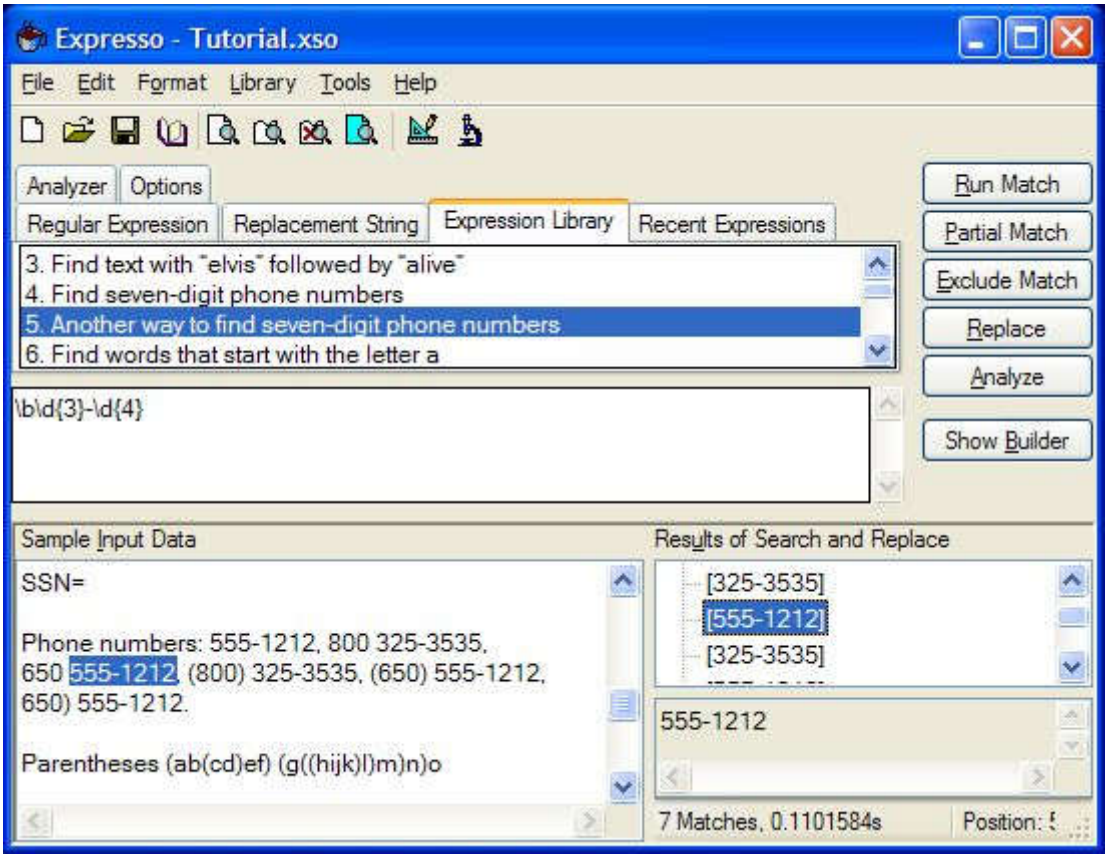


Figure 1. Expresso running example 5

Start by selecting the first example, "1. Find Elvis". Click Run Match and look at the TreeView on the right. Note there are several matches. Click on each to show the location of the match in the sample text. Run the second and third examples, noting that the word "pelvis" no longer matches. Finally, run the fourth and fifth examples; both should match the same numbers in the text. Try removing the initial "`\b`" and note that part of a Zip Code matched the format for a phone number.

Basics of .NET Regular Expressions

Let's explore some of the basics of regular expressions in .NET.

Special Characters

You should get to know a few characters with special meaning. You already met "`\b`", "`.`", "`*`", and "`\d`". To match any whitespace characters, like spaces, tabs, and newlines, use "`\s`" . Similarly, "`\w`" matches any alphanumeric character.

Let's try a few more examples:

6. `\ba\w*\b` *Find words that start with the letter a*

This works by searching for the beginning of a word (\b), then the letter "a", then any number of repetitions of alphanumeric characters (\w*), then the end of a word (\b).

7. `\d+` *Find repeated strings of digits*

Here, the "+" is similar to "*", except it requires at least one repetition.

8. `\b\w{6}\b` *Find six letter words*

Try these in Expresso and start experimenting by inventing your own expressions. Here is a table of some of the characters with special meaning:

- `.` Match any character except newline

`\w`	Match any alphanumeric character
`\s`	Match any whitespace character
`\d`	Match any digit
`\b`	Match the beginning or end of a word
`^`	Match the beginning of the string
`$`	Match the end of the string

Table 1. Commonly used special characters for regular expressions

In the beginning

The special characters "`^`" and "`$`" are used when looking for something that must start at the beginning of the text and/or end at the end of the text. This is especially useful for validating input in which the entire text must match a pattern. For example, to validate a seven-digit phone number, you might use:

9. `^\d{3}-\d{4}$` *Validate a seven-digit phone number*

This is the same as example (5), but forced to fill the whole text string, with nothing else before or after the matched text. By setting the "Multiline" option in .NET, "`^`" and "`$`" change their meaning to match the beginning and end of a single line of text, rather than the entire text string. The Expresso example uses this option.

Escaped characters

A problem occurs if you actually want to match one of the special characters, like "`^`" or "`$`". Use the backslash to remove the special meaning. Thus, "`\^`", "`\.`", and "`\\`", match the literal characters "`^`", "`.`", and "`\`", respectively.

Repetitions

You've seen that "`{3}`" and "`*`" can be used to indicate repetition of a single character. Later, you'll see how the same syntax can be used to repeat entire subexpressions. There are several other ways to specify a repetition, as shown in this table:

`*`	Repeat any number of times
`+`	Repeat one or more times
`?`	Repeat zero or one time
`{`*n*`}`	Repeat *n* times
`{`*n*`,`*m*`}`	Repeat at least *n*, but no more than *m* times
`{`*n*`,}`	Repeat at least *n* times

Table 2. Commonly used quantifiers

Let's try a few more examples:

10. `\b\w{5,6}\b` *Find all five and six letter words*

11. `\b\d{3}\s\d{3}-\d{4}` *Find ten digit phone numbers*

12. `\d{3}-\d{2}-\d{4}` *Social security number*

13. `^\w*` *The first word in the line or in the text*

Try the last example with and without setting the "Multiline" option, which changes the meaning of "`^`".

Character Classes

It is simple to find alphanumerics, digits, and whitespace, but what if we want to find anything from some other set of characters? This is easily done by listing the desired characters within square brackets. Thus, "`[aeiou]`" matches any vowel and "`[.?!]`" matches the punctuation at the end of a sentence. In this example, notice that the "`.`" And "`?`" lose their special meanings within square brackets and are interpreted literally. We can also specify a range of characters, so "`[a-z0-9]`" means, "match any lowercase letter of the alphabet, or any digit".

Let's try a more complicated expression that searches for telephone numbers.

14. `\(?\d{3}[) ]\s?\d{3}[- ]\d{4}` *A ten digit phone number*

This expression will find phone numbers in several formats, like "(800) 325-3535" or "650 555 1212". The "`\(?`" searches for zero or one left parentheses, "`[) ]`" searches for a right parenthesis or a space. The "\s?" searches for zero or one whitespace characters. Unfortunately, it will also find cases like "650) 555-1212" in which the parenthesis is not balanced. Below, you'll see how to use alternatives to eliminate this problem.

Negation

Sometimes we need to search for a character that is NOT a member of an easily defined class of characters. The following table shows how this can be specified.

`\W`	Match any character that is NOT alphanumeric
`\S`	Match any character that is NOT whitespace
`\D`	Match any character that is NOT a digit
`\B`	Match a position that is NOT the beginning or end of a word
`[^x]`	Match any character that is NOT *x*
`[^aeiou]`	Match any character that is NOT one of the characters *aeiou*

Table 3. How to specify what you don't want

15. `\S+` *All strings that do not contain whitespace characters*

Later, we'll see how to use "lookahead" and "lookbehind" to search for the absence of more complex patterns.

Alternatives

To select between several alternatives, allowing a match if either one is satisfied, use the pipe "`|`" symbol to separate the alternatives. For example, Zip Codes come in two flavors, one with 5 digits, the other with 9 digits and a hyphen. We can find either with this expression:

16. `\b\d{5}-\d{4}\b|\b\d{5}\b` *Five and nine digit Zip Codes*

When using alternatives, the order is important since the matching algorithm will attempt to match the leftmost alternative first. If the order is reversed in this example, the expression will only find the 5 digit Zip Codes and fail to find the 9 digit ones. We can use alternatives to improve the expression for ten digit phone numbers, allowing the area code to appear either delimited by whitespace or parenthesis:

17. `(\(\d{3}\)|\d{3})\s?\d{3}[- ]\d{4}` *Ten digit phone numbers, a better way*

Grouping

Parentheses may be used to delimit a subexpression to allow repetition or other special treatment. For example:

18. `(\d{1,3}\.){3}\d{1,3}` *A simple IP address finder*

The first part of the expression searches for a one to three digit number followed by a literal period "`\.`". This is enclosed in parentheses and repeated three times using the "`{3}`" quantifier, followed by the same expression without the trailing period.

Unfortunately, this example allows IP addresses with arbitrary one, two, or three digit numbers separated by periods even though a valid IP address cannot have numbers larger than 255. It would be nice to arithmetically compare a captured number N to enforce N<256, but this is not possible with regular expressions alone. The next example tests various alternatives based on the starting digits to guarantee the limited range of numbers by pattern matching. This shows that an expression can become cumbersome even when looking for a pattern that is simple to describe.

19. `((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)` *IP finder*

Expresso Analyzer View

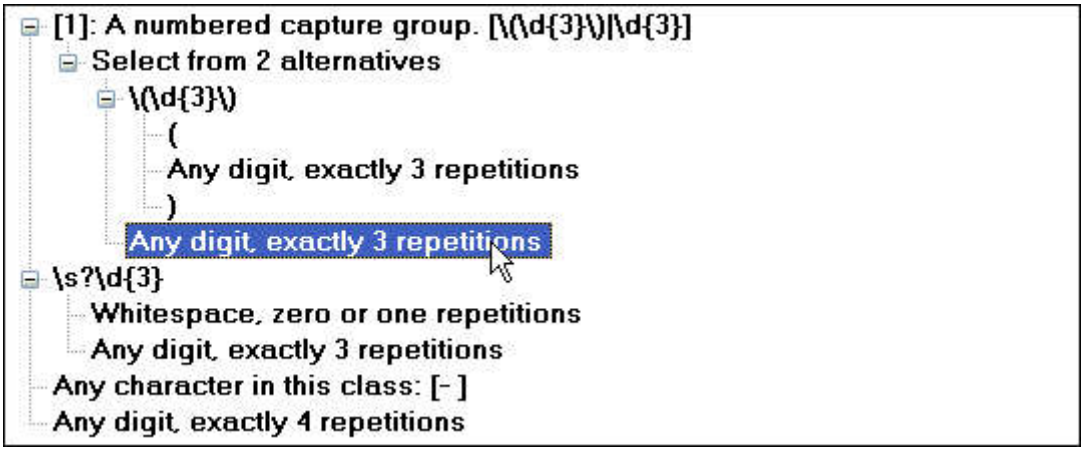


Figure 2. Expresso's analyzer view showing example 17

Expresso has a feature that diagrams expressions in a Tree structure, explaining what each piece means. When debugging an expression, this can help zoom in on the part that is causing trouble. Try this by selecting example (17) and then using the Analyze button. Select nodes in the tree and expand them to explore the structure of this regular expression as shown in the figure. After highlighting a node, you can also use the Partial Match or Exclude Match buttons to run a match using just the highlighted portion of the regular expression or using the regular expression with the highlighted portion excluded.

When subexpressions are grouped with parentheses, the text that matches the subexpression is available for further processing in a computer program or within the regular expression itself. By default, groups are numbered sequentially as encountered in reading from left to right, starting with 1. This automatic numbering can be seen in Expresso's skeleton view or in the results shown after a successful match.

A "backreference" is used to search for a recurrence of previously matched text that has been captured by a group. For example, "`\1`" means, "match the text that was captured by group 1". Here is an example:

20. `\b(\w+)\b\s*\1\b` *Find repeated words*

This works by capturing a string of at least one alphanumeric character within group 1 "`(\w+)`", but only if it begins and ends a word. It then looks for any amount of whitespace "`\s*`" followed by a repetition of the captured text "`\1`" ending at the end of a word.

It is possible to override the automatic numbering of groups by specifying an explicit name or number. In the above example, instead of writing the group as "`(\w+)`", we can write it as "`(?<Word>\w+)`" to name this capture group "`Word`". A backreference to this group is written "`\k<Word>`". Try this example:

21. `\b(?<Word>\w+)\b\s*\k<Word>\b` *Capture repeated word in a named group*

Test this in Expresso and expand the match results to see the contents of the named group.

Using parentheses, there are many special purpose syntax elements available. Some of the most common are summarized in this table:

Captures	
`(`*exp*`)`	Match *exp* and capture it in an automatically numbered group
`(?<`*name*`>`*exp*`)`	Match *exp* and capture it in a group named *name*
`(?:`*exp*`)`	Match *exp*, but do not capture it
Lookarounds	
`(?=`*exp*`)`	Match any position preceding a suffix *exp*
`(?<=`*exp*`)`	Match any position following a prefix *exp*
`(?!`*exp*`)`	Match any position after which the suffix *exp* is not found
`(?<!`*exp*`)`	Match any position before which the prefix *exp* is not found
Comment	
`(?#`*comment*`)`	Comment

Table 4. Commonly used Group Constructs

We've already talked about the first two. The third "`(?:`*exp*`)`" does not alter the matching behavior, it just doesn't capture it in a named or numbered group like the first two.

Positive Lookaround

The next four are so-called lookahead or lookbehind assertions. They look for things that go before or after the current match without including them in the match. It is important to understand that these expressions match a position like "`^`" or "`\b`" and never match any text. For this reason, they are known as "zero-width assertions". They are best illustrated by example:

"`(?=`*exp*`)`" is the "zero-width positive lookahead assertion". It matches a position in the text that precedes a given suffix, but doesn't include the suffix in the match:

22. `\b\w+(?=ing\b)` *The beginning of words ending with "ing"*

"`(?<=`*exp*`)`" is the "zero-width positive lookbehind assertion". It matches the position following a prefix, but doesn't include the prefix in the match:

23. `(?<=\bre)\w+\b` *The end of words starting with "re"*

Here is an example that could be used repeatedly to insert commas into numbers in groups of three digits:

24. `(?<=\d)\d{3}\b` *Three digits at the end of a word, preceded by a digit*

Here is an example that looks for both a prefix and a suffix:

25. `(?<=\s)\w+(?=\s)` *Alphanumeric strings bounded by whitespace*

Negative Lookaround

Earlier, I showed how to search for a character that is not a specific character or the member of a character class. What if we simply want to verify that a character is not present, but don't want to match anything? For example, what if we are searching for words in which the letter "q" is not followed by the letter "u"? We could try:

26. `\b\w*q[^u]\w*\b` *Words with "q" followed by NOT "u"*

Run the example and you will see that it fails when "q" is the last letter of a word, as in "Iraq". This is because "[^q]" always matches a character. If "q" is the last character of the word, it will match the whitespace character that follows, so in the example the expression ends up matching two whole words. Negative lookaround solves this problem because it matches a position and does not consume any text. As with positive lookaround, it can also be used to match the position of an arbitrarily complex subexpression, rather than just a single character. We can now do a better job:

27. `\b\w*q(?!u)\w*\b` *Search for words with "q" not followed by "u"*

We used the "zero-width negative lookahead assertion", "(?!*exp*)", which succeeds only if the suffix "*exp*" is not present. Here is another example:

28. `\d{3}(?!\d)` *Three digits not followed by another digit*

Similarly, we can use "**`(?<!`***exp***`)`**", the "zero-width negative lookbehind assertion", to search for a position in the text at which the prefix "*exp*" is not present:

29. `(?<![a-z ])\w{7}` *Strings of 7 alphanumerics not preceded by a letter or space*

Here is one more example using lookaround:

30. `(?<=<(\w+)>).*(?=<\/\1>)` *Text between HTML tags*

This searches for an HTML tag using lookbehind and the corresponding closing tag using lookahead, thus capturing the intervening text but excluding both tags.

Comments please

Another use of parentheses is to include comments using the "(?#comment)" syntax. A better method is to set the "Ignore Pattern Whitespace" option, which allows whitespace to be inserted in the expression and then ignored when the expression is used. With this option set, anything following a number sign "#" at the end of each line of text is ignored. For example, we can format the preceding example like this:

31. *Text between HTML tags, with comments*

```
(?<=    # Search for a prefix, but exclude it
  <(\w+)> # Match a tag of alphanumerics within angle brackets
)       # End the prefix
```

```
.* # Match any text
```

```
(?=     # Search for a suffix, but exclude it
  <\/\1>  # Match the previously captured tag preceded by "/"
)       # End the suffix
```

Greedy and Lazy

When a regular expression has a quantifier that can accept a range of repetitions (like "`.*`"), the normal behavior is to match as many characters as possible. Consider the following regular expression:

32. `a.*b` *The longest string starting with a and ending with b*

If this is used to search the string "aabab", it will match the entire string "aabab". This is called "greedy" matching. Sometimes, we prefer "lazy" matching in which a match using the minimum number of repetitions is found. All the quantifiers in Table 2 can be turned into "lazy" quantifiers by adding a question mark "`?`". Thus "`*?`" means "match any number of repetitions, but use the smallest number of repetitions that still leads to a successful match". Now let's try the lazy version of example (32):

33. `a.*?b` *The shortest string starting with a and ending with b*

If we apply this to the same string "aabab" it will first match "aab" and then "ab".

`*?`	Repeat any number of times, but as few as possible
`+?`	Repeat one or more times, but as few as possible
`??`	Repeat zero or one time, but as few as possible
`{`*n*`,`*m*`}?`	Repeat at least *n*, but no more than *m* times, but as few as possible
`{`*n*`,}?`	Repeat at least *n* times, but as few as possible

Table 5. Lazy quantifiers

What did we leave out?

I've described a rich set of elements with which to begin building regular expressions; but I left out a few things that are summarized in the following table. Many of these are illustrated with additional examples in the project file. The example number is shown in the left-hand column of this table.

#	Syntax	Description
	`\a`	Bell character
	`\b`	Normally a word boundary, but within a character class it means backspace
	`\t`	Tab
34	`\r`	Carriage return
	`\v`	Vertical tab
	`\f`	Form feed
35	`\n`	New line
	`\e`	Escape
36	`\`*nnn*	Character whose ASCII octal code is *nnn*
37	`\x`*nn*	Character whose hexadecimal code is *nn*
38	`\u`*nnnn*	Character whose Unicode is *nnnn*

39	**\c***N*	Control *N* character, for example carriage return (Ctrl-M) is \cM
40	\A	Beginning of a string (like ^ but does not depend on the multiline option)
41	\Z	End of string or before \n at end of string (ignores multiline)
	\z	End of string (ignores multiline)
42	\G	Beginning of the current search
43	**\p{***name***}**	Any character from the Unicode class named *name*, for example \p{IsGreek}
	(?>*exp***)**	Greedy subexpression, also known as a non-backtracking subexpression. This is matched only once and then does not participate in backtracking.
44	**(?<***x***>-<***y***>***exp***)** or **(?-<***y***>***exp***)**	Balancing group. This is complicated but powerful. It allows named capture groups to be manipulated on a push down/pop up stack and can be used, for example, to search for matching parentheses, which is otherwise not possible with regular expressions. See the example in the project file.
45	**(?im-nsx:***exp***)**	Change the regular expression options for the subexpression *exp*
46	(?im-nsx)	Change the regular expression options for the rest of the enclosing group
	(?(*exp***)***yes***|***no***)**	The subexpression *exp* is treated as a zero-width positive lookahead. If it matches at this point, the subexpression *yes* becomes the next match, otherwise *no* is used.
	(?(*exp***)***yes***)**	Same as above but with an empty *no* expression
	(?(*name***)***yes***|***no***)**	This is the same syntax as the preceding case. If name is a valid group name, the *yes* expression is matched if the named group had a successful match, otherwise the *no* expression is matched.
47	**(?(***name***)***yes***)**	Same as above but with an empty *no* expression

Table 6. Everything we left out. The left-hand column shows the number of an example in the project file that illustrates this construct.

Conclusion

We've given many examples to illustrate the essential features of .NET regular expressions, emphasizing the use of a tool like Expresso to test, experiment, and learn by example. If you get hooked, there are many online resources available to help you go further. You can start your search at the Ultrapico web site. If you want to read a book, I suggest the latest edition of *Mastering Regular Expressions,* by Jeffrey Friedl.

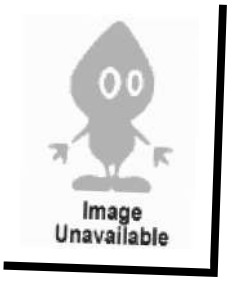
There are also a number of nice articles on The Code Project including the following tutorials:

- An Introduction to Regular Expressions by Uwe Keim
- Microsoft Visual C# .NET Developer's Cookbook: Chapter on Strings and Regular Expressions

License

Share

About the Author



Jim Hollenhorst

Researcher

United States

Ultrapico Website: http://www.ultrapico.com

Download Expresso 3.0, the latest version of the award-winning regular expression development tool.

You may also be interested in...

- Delegates - a 15 minutes quick start tutorial
- Internet of Things Security from the Ground Up
- Improving ASP.NET MVC MUSIC STORE more usability with DotNetAge in 30 minutes
- Build BOT with Microsoft Bot Framework Rest API
- Visual COBOL New Release: Small point. Big deal
- SAPrefs - Netscape-like Preferences Dialog

Comments and Discussions

505 messages have been posted for this article Visit **http://www.codeproject.com/Articles/9099/The-Minute-Regex-Tutorial** to post and view comments on this article, or click **here** to get a print view with messages.