ZYTRAX.COM info

Regular Expressions - User Guide

A Regular Expression is the term used to describe a codified method of searching *invented*, or *defined*, by the American mathematician Stephen Kleene.

The syntax (language format) described on this page is compliant with **extended regular expressions (EREs)** defined in IEEE POSIX 1003.2 (Section 2.8). **EREs** are now commonly supported by Apache, PERL, PHP4, Javascript 1.3+, MS Visual Studio, most visual editors, vi, emac, the GNU family of tools (including grep, awk and sed) as well as many others. **Extended Regular Expressions (EREs)** will support **Basic Regular Expressions** (BREs are essentially a subset of EREs). Most applications, utilities and laguages that implement RE's, especially PERL, extend the capabilities defined and this has become, mostly, the *de facto* standard. The appropriate documentation should always be consulted.

Translation: The page has been translated into Bulgarian, courtesy of Albert Ward - thanks.

Contents

A Gentle Introduction: - the Basics
Simple Searches
Brackets, Ranges and Negation [] , - and ^
Search Positioning (aka Anchors) ^ and $
Iteration (aka Quantifiers) ?, *, + , {n}, {n.m} and {n,}
Parenthesis and Alternation () and |
POSIX Standard Character Classes:
Commonly Available extensions: - \w etc
Subexpressions, Submatches, Groups and Backreferences:
Regular Expression Tester: - Experiment with your own target strings and search expressions in your browser
Some Examples: - A worked example and some samples
Notes: - general notes when using utilities and lanuages
Utility notes: - using Visual Studio regular expressions
Utility notes: - using sed for file manipulation (not for the faint hearted)

A Gentle Introduction: The Basics

The title is deceptive. There is no gentle beginning to regular expressions. You are either into hieroglyphics big time - in which case you will love this stuff - or you need to use regular expression, in which case your only reward may be a headache. But they are jolly useful. Sometimes.

Some Definitions before we start

We are going to be using the terms **literal**, **metacharacter**, **target string**, **escape sequence** and **search expression** (aka regular expression) in this overview. Here is a definition of our terms:

literal	A **literal** is any character we use in a search or matching expression, for example, to find **ind** in w**ind**ows the **ind** is a **literal** string - each character plays a part in the search, it is

	literally the string we want to find.
metacharacter	A **metacharacter** is one or more special characters that have a unique meaning and are NOT used as **literals** in the search expression, for example, the character ^ (circumflex or caret) is a **metacharacter**.
target string	This term describes the string that we will be searching, that is, the string in which we want to find our match or search pattern.
search expression	Most commonly called the regular expression. This term describes the search expression that we will be using to search our target string, that is, the pattern we use to find what we want.
escape sequence	An **escape sequence** is a way of indicating that we want to use one of our **metacharacters** as a **literal**. In a regular expression an **escape sequence** involves placing the **metacharacter** \ (backslash) in front of the **metacharacter** that we want to use as a **literal**, for example, if we want to find **(s)** in the target string **window(s)** then we use the search expression **\(s\)** and if we want to find **\\file** in the target string **c:\\file** then we would need to use the search expression **\\\\file** (each \ we want to search for as a **literal** (there are 2) is preceded by an **escape sequence** \).

Our Example Target Strings

Throughout this guide we will use the following as our target strings:

```
STRING1   Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)
STRING2   Mozilla/4.75 [en](X11;U;Linux2.2.16-22 i586)
```

These are Browser ID Strings and appear as the Apache Environmental variable HTTP_USER_AGENT (full list of Apache environmental variables).

Simple Matching

We are going to try some simple matching against our example target strings:

Note: You can also experiment as you go through the examples.

Search for (search expression)		
m	**STRING1** match	Finds the **m** in co**m**patible
	STRING2 no match	There is no lower case **m** in this string. Searches are case sensitive unless you take special action.
a/4	**STRING1** match	Found in Mozill**a/4**.0 - any combination of characters can be used for the match
	STRING2 match	Found in same place as in STRING1
5 [	**STRING1** no match	The search is looking for a pattern of '5 [' and this does NOT exist in STRING1. Spaces are valid in searches.

	STRING2	match	Found in Mozilla/4.7**5 [**en]
in	**STRING1**	match	found in W**in**dows
	STRING2	match	Found in L**in**ux
le	**STRING1**	match	found in compatib**le**
	STRING2	no match	There is an l and an e in this string but they are not adjacent (or contiguous).

Check the results in our Regular Expression Tester.

Brackets, Ranges and Negation

Bracket expressions introduce our first **metacharacters**, in this case the square brackets which allow us to define list of things to test for rather than the single characters we have been checking up until now. These lists can be grouped into what are known as Character Classes typically comprising well know groups such as all numbers etc.

Metacharacter	Meaning
[ ]	Match anything inside the square brackets for ONE character position once and only once, for example, [12] means match the target to 1 and if that does not match then match the target to 2 while [0123456789] means match to any character in the range 0 to 9.
-	The - (dash) **inside square brackets** is the 'range separator' and allows us to define a range, in our example above of [0123456789] we could rewrite it as [0-9]. You can define more than one range inside a list, for example, [0-9A-C] means check for 0 to 9 and A to C (but not a to c). **NOTE:** To test for - inside brackets (as a **literal**) it must come first or last, that is, [-0-9] will test for - and 0 to 9.
^	The ^ (circumflex or caret) **inside square brackets** negates the expression (we will see an alternate use for the circumflex/caret **outside** square brackets later), for example, [^Ff] means anything except upper or lower case F and [^a-z] means everything except lower case a to z. **Notes:** 1. There are no spaces between the range delimiter values, if there was, depending on the range, it would be added to the possible range or rejected as invalid. Be very careful with spaces. 2. Some regular expression systems, notably VBScript, provide a negation operator (!) for use with strings. This is a non-standard feature and therefore the resulting expressions are not portable. 3. Negation can be very tricky - you may want to read these

additional notes on this and other topics.

NOTE: There are some special range values (Character Classes) that are built-in to most regular expression software and have to be if it claims POSIX 1003.2 compliance for either BRE or ERE.

So let's try this new stuff with our target strings.

Search for (search expression)			
in[du]	**STRING1**	match	finds ind in W**ind**ows
	STRING2	match	finds **inu** in L**inu**x
x[0-9A-Z]	**STRING1**	no match	Again the tests are case sensitive to find the xt in DigE**xt** we would need to use [0-9a-z] or [0-9A-Zt]. We can also use this format for testing upper and lower case e.g. [Ff] will check for lower and upper case F.
	STRING2	match	Finds x2 in Linu**x2**
[^A-M]in	**STRING1**	match	Finds Win in **Win**dows
	STRING2	no match	We have excluded the range A to M in our search so Linux is not found but linux (if it were present) would be found.

Check the results in our Regular Expression Tester.

Positioning (or Anchors)

We can control where in our target strings the matches are valid. The following is a list of **metacharacters** that affect the position of the search:

Metacharacter	Meaning
^	The ^ (circumflex or caret) **outside square brackets** means look only at the beginning of the target string, for example, **^Win** will not find Windows in **STRING1** but **^Moz** will find **Moz**illa.
$	The $ (dollar) means look only at the end of the target string, for example, fox$ will find a match in 'silver **fox**' since it appears at the end of the string but not in 'the fox jumped over the moon'.
.	The . (period) means any character(s) in this position, for example, **ton.** will find **tons**, **tone** and **tonneau** but not **wanton** because it has no following character.

NOTE: Many systems and utilities, but not all, support special positioning macros, for example \< match at beginning of word, \> match at end of word, \b match at the begining OR end of word , \B except at the beginning or end of a word. List of the common values.

So let's try this lot out with our example target strings..

Search for

(search expression)			
[a-z]\)$	**STRING1**	match	finds t) in DigiEx**t) Note:** The \ is an escape character and is required to treat the) as a literal
	STRING2	no match	We have a numeric value at the end of this string but we would need [0-9a-z]) to find it.
.in	**STRING1**	match	Finds **Win** in **Win**dows.
	STRING2	match	Finds Lin in **Lin**ux.

Check the results in our Regular Expression Tester.

Iteration 'metacharacters'

The following is a set of **iteration metacharacters** (a.k.a. quantifiers) that can control the number of times the preceding **character** is found in our searches. The iteration meta characters can also be used in conjunction with parenthesis meta characters.

Metacharacter	Meaning
?	The ? (question mark) matches when the preceding character occurs 0 or 1 times only, for example, colou?r will find both color (u is found 0 times) and colour (u is found 1 time).
*	The * (asterisk or star) matches when the preceding character occurs 0 or more times, for example, tre* will find tree (e is found 2 times) and tread (e is found 1 time) and trough (e is found 0 times).
+	The + (plus) matches when the preceding character occurs 1 or more times, for example, tre+ will find tree (e is found 2 times) and tread (e is found 1 time) but NOT trough (0 times).
{n}	Matches when the preceding character, or character range, occurs n times exactly, for example, to find a local phone number we could use **[0-9]{3}-[0-9]{4}** which would find any number of the form 123-4567. Value is enclosed in braces (curly brackets). **Note:** The - (dash) in this case, because it is outside the square brackets, is a **literal**. Louise Rains writes to say that it is invalid to commence a NXX code (the **123**) with a zero (which would be permitted in the expression above). In this case the expression **[1-9][0-9]{2}-[0-9]{4}** would be necessary to find a valid local phone number.
{n,m}	Matches when the preceding character occurs at least n times but not more than m times, for example, **ba{2,3}b** will find **baab** and **baaab** but NOT **bab** or **baaaab**. Values are enclosed in braces (curly brackets).
{n,}	Matches when the preceding character occurs at least n times, for example, **ba{2,}b** will find 'baab', 'baaab' or 'baaaab' but NOT 'bab'. Values are enclosed in braces (curly brackets).

Note: While it may be obvious to some, it is also worth emphasizing what characters play

a role in iteration. In all the above examples **only the character immediately preceding the iteration character or expression** takes part in the iteration, all other characters in the search expression (regular expression) are **literals**. Thus, in the first example search expression **colou?r**, the string **colo** is a literal and must be found before the iteration sequence (**u?**) is triggered which, if satisfied, must also be followed by the literal **r** for a match to occur.

So let's try them out with our example target strings.

Search for (search expression)			
\(.*l	**STRING1**	match	finds the (and l in **(**compatib**l**e. The opening \ is an escape character used to indicate the (it precedes is a literal (search character) not a metacharacter. **Note:** If you use the tester with **STRING1** and the above expression it will return the match **(compatibl**. The literal (essentially anchors the search - it simply says start the search only when an (is found. The following **.*** says the (may be followed by any character (.), zero or more times (*) (thus **compatib** are essentially random characters that happen to appear in this string - they were not part of the search) and terminate the search on finding an **l** literal. Only the **(** and **l** are truly part of the search expression.
	STRING2	no match	Mozilla contains lls but not preceded by an open parenthesis (no match) and Linux has an upper case L (no match).

We had previously defined the above test using the search value **l?** (thanks to David Werner Wiebe for pointing out our error). The search expression l? actually means find anything, even if it has no l (l 0 or 1 times), so would match on both strings. We had been looking for a method to find a single l and exclude ll which, without lookahead (a relatively new extension to regular expressions pioneered by PERL) is pretty difficult. Well, that is our excuse.

W*in	**STRING1**	match	Finds the Win in **Win**dows.
	STRING2	match	Finds **in** in L**in**ux preceded by W zero times - so a match.
[xX][0-9a-z]{2}	**STRING1**	no match	Finds x in DigExt but only one t.
	STRING2	match	Finds X and 11 in X11.

Check the results in the Regular Expression Tester.

More 'metacharacters'

The following is a set of additional **metacharacters** that provide added power to our searches:

Metacharacter	Meaning
()	The ((open parenthesis) and) (close parenthesis) may be used to group (or bind) parts of our search expression together. Officially this is called a subexpression (a.k.a. a submatch or group) and subexpressions may be nested to any depth. Parentheses (subexpresions) also capture the matched element into a variable that may be used as a backreference. See this example for its use in binding OR more about subexpressions (aka grouping or submatching) and their use as backreferences.
|	The | (vertical bar or pipe) is called **alternation** in techspeak and means find the left hand OR right values, for example, **gr(a|e)y** will find 'gray' or 'grey' and has the sense that - having found the literal characters 'gr' - if the first test is not valid (a) the second will be tried (e), if the first is valid the second will not be tried. Alternation can be nested within each expression, thus **gr((a|e)|i)y** will find 'gray', 'grey' and 'griy'.

<humblepie> In our examples, we blew this expression ^([L-Z]in), we incorrectly stated that this would negate the tests [L-Z], the '^' only performs this function **inside** square brackets, here it is **outside** the square brackets and is an **anchor** indicating 'start from first character'. Many thanks to Mirko Stojanovic for pointing it out and apologies to one and all.**</humblepie>**

So let's try these out with our example strings.

Search for (search expression)			
^([L-Z]in)	**STRING1**	no match	The '^' is an anchor (because it lies outside any square brackets) indicating first position. **Win** does not start the string so no match.
	STRING2	no match	The '^' is an anchor (because it lies outside any square brackets) indicating first position. **Lin**ux does not start the string so no match.
((4\.[0-3])|(2\.[0-3]))	**STRING1**	match	Finds the **4.0** in Mozilla/**4.0**. The '\.' sequence uses the escape metacharacter (\) to ensure that the '.' (dot) is used as a literal in the search.
	STRING2	match	Finds the **2.2** in Linux**2.2**.16-22.
(W|L)in	**STRING1**	match	Finds **Win** in **Win**dows.
	STRING2	match	Finds **Lin** in **Lin**ux.

Check the results in the Regular Expression Tester.

More Stuff

Contents

POSIX Standard Character Classes
Apache browser recognition - a worked example
Commonly Available extensions - \w etc
Submatches, Groups and Backreferences
Regular Expression Tester - Experiment with your own strings and expressions in your browser
Common examples - regular expression examples
Notes - general notes when using utilities and lanuages
Utility notes - using Visual Studio regular expressions
Utility notes - using sed for file manipulation (not for the faint hearted)

For more information on regular expressions go to our links pages under Languages/regex. There are lots of folks who get a real buzz out of making any search a 'one liner' and they are incredibly helpful at telling you how they did it. Welcome to the wonderful, if arcane, world of Regular Expressions. You may want to play around with your new found knowledge using this tool.

POSIX Character Class Definitions

POSIX 1003.2 section 2.8.3.2 (6) defines a set of character classes that denote certain common ranges. They tend to look very ugly but have the advantage that also take into account the 'locale', that is, any variant of the local language/coding system. Many utilities/languages provide short-hand ways of invoking these classes. Strictly the names used and hence their contents reference the LC_CTYPE POSIX definition (1003.2 section 2.5.2.1).

Value	Meaning
[:digit:]	Only the digits 0 to 9
[:alnum:]	Any alphanumeric character 0 to 9 OR A to Z or a to z.
[:alpha:]	Any alpha character A to Z or a to z.
[:blank:]	Space and TAB characters only.
[:xdigit:]	Hexadecimal notation 0-9, A-F, a-f.
[:punct:]	Punctuation symbols . , " ' ? ! ; : # $ % & ( ) * + - / < > = @ [ ] \ ^ _ { } | ~
[:print:]	Any printable character.
[:space:]	Any whitespace characters (space, tab, NL, FF, VT, CR). Many system abbreviate as \s.
[:graph:]	Exclude whitespace (SPACE, TAB). Many system abbreviate as \W.
[:upper:]	Any alpha character A to Z.

[:lower:]	Any alpha character a to z.
[:cntrl:]	Control Characters NL CR LF TAB VT FF NUL SOH STX EXT EOT ENQ ACK SO SI DLE DC1 DC2 DC3 DC4 NAK SYN ETB CAN EM SUB ESC IS1 IS2 IS3 IS4 DEL.

These are always used **inside square brackets** in the form [[:alnum:]] or combined as [[:digit:]a-d]

Common Extensions and Abbreviations

Some utitlities and most languages provide extensions or abbreviations to simplify(!) regular expressions. These tend to fall into Character Classes or position extensions and the most common are listed below. In general these extensions are defined by PERL and implemented in what is called PCRE's (Perl Compatible Regular Expressions) which has been implemented in the form of a libary that has been ported to many systems. Full details of PCRE. PERL 5.8.8 regular expression documentation.

While the \x type syntax for can look initially confusing the backslash precedes a character that does not normally need escaping and hence can be interpreted correctly by the utility or language - whereas we simple humans tend to become confused more easily. The following are supported by: .NET, PHP, PERL, RUBY, PYTHON, Javascript as well as many others.

Character Class Abbreviations	
\d	Match any character in the range 0 - 9 (equivalent of POSIX [:digit:])
\D	Match any character NOT in the range 0 - 9 (equivalent of POSIX [^[:digit:]])
\s	Match any whitespace characters (space, tab etc.). (equivalent of POSIX [:space:] EXCEPT VT is not recognized)
\S	Match any character NOT whitespace (space, tab). (equivalent of POSIX [^[:space:]])
\w	Match any character in the range 0 - 9, A - Z and a - z (equivalent of POSIX [:alnum:])
\W	Match any character NOT the range 0 - 9, A - Z and a - z (equivalent of POSIX [^[:alnum:]])
Positional Abbreviations	
\b	Word boundary. Match any character(s) at the beginning (\bxx) and/or end (xx\b) of a word, thus \bton\b will find ton but not tons, but \bton will find tons.
\B	Not word boundary. Match any character(s) NOT at the beginning(\Bxx) and/or end (xx\B) of a word, thus \Bton\B will find wantons but not tons, but ton\B will find both wantons and tons.

Subexpressions, Submatches, Groups and Backreferences

All regular expression implementations that claim BRE (or higher) compatibility provide the last results of each separate match enclosed in parenthesis (officially called a **subexpression** but frequently called a **submatch** or group) in **variables** that may subsequently (after the regular expression has been executed) be used or substituted in an expression by using a **backreference**. There may be one or more such groupings in any regular expression. These variables are usually numbered $1 to $9. Where $1 will contain the first **submatch**, $2 will contain the second **submatch** and so on. The $x value typically persists until another regular expression is encountered. Examples:

```
# assume target string = "cat"
search expression = (c|a)(t|z)
$1 will contain "a"
# the 'c' is found but the next character fails (t|z)
# the search advances by one character
# the 'c' is not found but 'a' is and the
# next character finds 't' in (t|z) - match
# if the target string was "act"
# $1 would contain "c"
$2 will contain "t" in both cases

# OpenLDAP 'access to' directive example: assume target dn
# is "ou=something,cn=my name,dc=example,dc=com"
# then $1 = 'my name' at end of match below
# because first regular expression does not use ()
access to dn.regex="ou=[^,]+,cn=([^,]+),dc=example,dc=com"
# subsequent expression contains backreference
 by dn.exact,expand="cn=$1,dc=example,dc=com"

# However, in the following directive
access to dn.regex="ou=([^,]+),cn=([^,]+),dc=example,dc=com"
# subsequent expression contains backreference
 by dn.exact,expand="cn=$2,dc=example,dc=com"
# $1 will contain 'something' and
# $2 will contain 'my name' because
# both regular expressions use ()
```

When used within a single expression these submatches (subexpressions) are typically called groups and are placed in numeric variables addressed using a **backreference** of the form \1 to \9. These groups or backreferences (variables) may be substituted within the regular expression. The following example demonstrates usage:

```
# the following expression finds any occurrence of double characters
(.)\1
# the parenthesis creates the grouping (or submatch or subexpression)
# in this case it is the first (only), so is backreferenced by \1
# the . (dot) finds any character and the \1 backrefence substitutes whatever
# character was found by the dot in the next character position,
# thus to match it must find two consecutive characters which are the same
```

It is possible to suppress the capture of any subexpression or group (enclosed in parenthesis) into a backreference by adding the string '?:' immediately after the opening parenthesis (. The following example illustrates this behaviour:

```
# in both cases assume target dn
# is "ou=something,cn=my name,dc=example,dc=com"

# all groups capture to backreference variables
access to dn.regex="ou=([^,]+),cn=([^,]+),dc=example,dc=com"
# subsequent expression contains backreference
 by dn.exact,expand="cn=$2,dc=example,dc=com"
# $1 will contain 'something' and
# $2 will contain 'my name' because
# both regular expressions use ()

# first group capture to backreference variable is suppressed
access to dn.regex="ou=(?:[^,]+),cn=([^,]+),dc=example,dc=com"
# subsequent expression contains backreference
 by dn.exact,expand="cn=$1,dc=example,dc=com"
# $1 will contain 'my name'
```

Regular Expression - Experiments and Testing

This simple regular expression tester lets you experiment using your browser's regular expression Javascript function (use View Source in your browser for the Javascript source code).

Enter or copy/paste the string you want to search in the box labled **String:** and the regular expression in the box labeled **RE:**, click the **Search** button and results will appear in the box labeled **Results:**. If you are very lucky the results may even be what you expect. This tester displays the whole searched string in the **Results** field and encloses in < > the first found result. This may not be terribly helpful if you are dealing with HTML - but our heart is in the right place. All matches are then displayed separately showing the found text and its character position in the string. Checking the **Case Insensitive:** box makes the search case insensitive thus [AZ] will find the "a" in "cat", whereas without checking the box [aZ] would be required to find the "a" in "cat". **Note:** Not all regular expression systems provide a case insensitivity feature and therefore the regular expression may not be portable. Checking **Results only** will supress display of the marked up original string and only show the results found, undoing all our helpful work, but which can make things a little less complicated especially if dealing with HTML strings or anything else with multiple < > symbols. **Clear** will zap all the fields - including the regular expression that you just took 6 hours to develop. **Use with care.** See the notes below for limitations, support and capabilities.

Notes:

1. If the regular expression is invalid or syntactically incorrect the tester will display in the **Results** field exactly what your browser thought of your attempt - sometimes it might even be useful.

2. The **Results** field now shows (assuming your syntax is valid) the value of any backreference variables (in the range $1 - $9). These backreferences can either be accessed by the use of the variables $1 to $9 in subsequent expressions or by \1 to \9 within the same regular expression. If none exist (there were no subexpressions or groups in the regular expression) then, surprisingly, none will be displayed.

3. **Backreferences:** displays all group in the regular expression in the order in which they were encountered and wheter or not they triggered termination. Thus, if a multiple level of nesting is used as in the string "cat" with a regular expression of **((c|a)|{a|t})** then 3 backreferences are displayed $1= c (the outer parenthesis was the first encountered and will always containing one of the match strings in the nested groups), $2= c (this is (c|a) group which causes the expression to terminate) and $3 = undefined (covers the (a|t) group which was not triggered in terminating the expression).

<ouch> We had an error such that if the match occurred in the first position the enclosing <> was incorrectly displayed.**</ouch>**

If you plan to experiment with the target strings used to illustrate usage of the various meta characters we have thoughtfully replicated them below to save you all that scrolling. Just copy and paste into the **String** box. Are we helpful or not?

```
STRING1   Mozilla/4.0 (compatible; MSIE 5.0; Windows NT; DigExt)
STRING2   Mozilla/4.75 [en](X11;U;Linux2.2.16-22 i586)
```

String:

RE:

Options: **Case Insensitive:** **Results Only:**

Clear Search

Results:

Notes:

1. Javascript implementations may vary from browser to browser. This feature was tested with MSIE 6.x, Gecko (Firefox 2.something) and Opera 9 (email indicates it works in Google's Chrome - so will likely work with any WebKit based browser, which obviously includes Safari (and now even Opera!)). If the tester does not work for you we are very, very sad - but yell at your browser supplier not us.

2. The ECMA-262 (Javascript 1.2'ish) spec defines the regex implementation to be based on Perl 5 which means that submatches/backreferences and short forms, such as \d, should be supported in addition to standard BRE and ERE functionality.

3. In Opera and MSIE the following backreference/submatch worked:

   ```
   (.)\1
   ```

   Which finds any occurence of double characters, such as, oo in spoon. Using Gecko (Firefox 2.something) it did not work. Since, at least, Gecko/20100228 (possibly even before that) the expression now works.

4. The regular expression tester terminates on the first match found and marks its position in the search string by enclosing it in <>, displays the whole match string together with character position where it started and any backeferences applicable at that point. The tester then iterates through the rest of the string to find further matches and displays only the match string together with the character position of the first character of the match string under the heading **Additional matches:**. Each iteration starts from the character immediately after the last character of any match string. Overlapped matches will not be found.

5. If you get the message **Match(es) = zero length** in the results field this implies your browser's Javascript engine has choked on executing the regular expression. It has returned a valid match (there may be others in the string) but has (incorrectly) returned a length of 0 for the number of characters in the match. This appears to

happen (Moz/FF and IE tested) when searching for a single character in a string when using the meta characters ? and *. For example, the regular expression **e?** on any string will generate the error (this expression will yield a match on every character in the string and is probably not what the user wanted).

6. For those of you familiar with Javascript's regular expressions there is no need to add the enclosing //, just enter the raw regular expression. Those of you not familiar with Javascript regular expressions - ignore the preceding sentence.

Some Examples

The following sections show a number of worked examples which may help to clarify regular expression. Most likely they will not.

Apache Browser Identification - a Worked Example

All we ever wanted to do with Regular Expressions was to find enough about visiting browsers arriving at our Apache powered web site to decide what HTML/CSS to supply or not for our pop-out menus. The Apache **BrowserMatch** directives will set a variable if the expression matches the USER_AGENT string.

We want to know:

- If we have any browser that supports Javascript (isJS).
- If we have any browser that supports the MSIE DHTML Object Model (isIE).
- If we have any browser that supports the W3C DOM (isW3C).

Here in their glory are the Apache regular expression statements we used (maybe you can understand them now)

```
BrowserMatchNoCase [Mm]ozilla/[4-6] isJS
BrowserMatchNoCase MSIE isIE
BrowserMatchNoCase [Gg]ecko isW3C
BrowserMatchNoCase MSIE.((5\.[5-9])|([6-9]|1[0-9])) isW3C
BrowserMatchNoCase W3C_ isW3C
```

Notes:

- Line 1 checks for any upper or lower case variant of Mozilla/4-6 (MSIE also sets this value). This test sets the variable isJS for all version 4-6 browsers (we assume that version 3 and lower do not support Javascript or at least not a sensible Javascript).
- Line 2 checks for MSIE only (line 1 will take out any MSIE 1-3 browsers even if this variable is set.
- Line 3 checks for any upper or lower case variant of the Gecko browser which includes Firefox, Netscape 6, 7 and now 8 and the Moz clones (all of which are Mozilla/5).

- Line 4 checks for MSIE 5.5 (or greater) OR MSIE 6 - 19 (future proofing - though at the rate MS is updating MSIE it will probably be out-of-date next month).
  NOTE about binding: This expression does not work:

```
BrowserMatchNoCase MSIE.(5\.[5-9])|([6-9]) isW3C
```

It incorrectly sets variable isW3C if the number 6 - 9 appears in the string. Our guess is the binding of the first parenthesis is directly to the MSIE expression and the OR and second parenthesis is treated as a separate expression. Adding the inner parenthesis fixed the problem.

- Line 5 checks for W3C_ in any part of the line. This allows us to identify the W3C validation services (either CSS or HTML/XHTML page validation).

Some of the above checks may be a bit excessive, for example, is Mozilla ever spelled mozilla?, but it is also pretty silly to have code fail just because of this 'easy to prevent' condition. There is apparently no final consensus that all Gecko browsers will have to use Gecko in their 'user-agent' string but it would be extremely foolish not to since this would force guys like us to make huge numbers of tests for branded products and the more likely outcome would be that we would not.

Common Examples

The following examples may be useful, they are particularly aimed at extracting parameters but cover some other ground as well. If anyone wants to email us some more examples we'd be happy to post with an appropriate credit.

```
# split on simple space
string "aaa bbb ccc"
re = \S+
result = "aaa", "bbb", "ccc"
# Note: If you want the location of the whitespace (space) use \s+

# css definition split on space or comma but keep "" enclosed items
string = '10pt "Times Roman",Helvetica,Arial,sans-serif'
re = \s*("[^"]+"|[^ ,]+)
result = "10pt", "\"Times Roman\"", "Helvetica", "Arial", "sans-serif"

# extract HTML <> enclosed tags
string = '<a href="#">A link</a>'
re = <[^>]*>
result = '<a href="#">', "</a>"

# find all double characters
string = 'aabcdde'
re = (.)\1
result = "aa", "dd"

# separate comma delimted values into groups (submatches or backreferences)
string = ou=people,cn=web,dc=example,dc=com
re = ou=[^,]+,cn=([^,]+),dc=example,dc=com
result $1 variable will contain "web" - first expression has no grouping ()
However, using
re = ou=([^,]+),cn=([^,]+),dc=example,dc=com
$1 will contain "people" and $2 will contain "web"
```

Utility and Language Notes - General

1. Certain utilities, notably grep, suggest that it is a good idea to enclose any complex search expression inside single quotes. In fact it is not a good idea - it is absolutely

essential! Example:

```
grep 'string\\' *.txt # this works correctly
grep string\\ *.txt # this does not work
```

2. Some utilities and most languages use / (forward slash) to start and end (de-limit or contain) the search expression others may use single quotes. This is especially true when there may be optional following arguments (see the grep example above). These characters do not play any role in the search itself.

Utility Notes - Using Visual Studio

For reasons best know to itself MS Visual Studio (VS.NET) uses a bizarre set of extensions to regular expressions. (MS VS standard documentation) But there is a free regular expression add-in if you want to return to sanity.

Utility Notes - Using sed

Stream editor (sed) is one of those amazingly powerful tools for manipulating files that are simply horrible when you try to use them - unless you get a buzz out of ancient Egyptian hieroglyphics. But well worth the effort. So if you are hieroglyphically-challenged, like us, these notes may help. There again they may not. There is also a useful series of tutorials on sed and this list of sed one liners.

1. **not all seds are equal:** Linux uses GNU sed, the BSDs use their own, slightly different, version.
2. **sed on windows:** GNU sed has been ported to windows.
3. **sed is line oriented:** sed operates on lines of text within the file or input stream.
4. **expression quoting:** To avoid shell expansion (in BASH especially) quote all expressions in single quotes as in a 'search expression'.
5. **sed defaults to BRE:** The default behaviour of sed is to support Basic Regular Expressions (BRE). To use all the features described on this page set the -r (Linux) or -E (BSD) flag to use Extended Regular Expressions (ERE) as shown:

```
# use ERE on Linux (GNU sed)
sed -r 'search expression' file
# use ERE on BSD
sed -E 'search expression' file
```

6. **in-situ editing:** By default sed outputs to 'Standard Out' (normally the console/shell). There are two mutually exclusive options to create modified files. Redirect 'standard out' to a file or use in-situ editing with the -i option. The following two lines illustrate the options:

```
# in-situ: saves the unmodified file to file.bak BEFORE
# modifying
sed -i .bak 'search expression' file

# redirection: file is UNCHANGED the modified file is file.bak
```

```
sed 'search expression' file > file.bak
```

7. **sed source:** Sed will read from a file or 'Standard In' and therefore may be used in piped sequences. The following two lines are functionally equivalent:

```
cat file |sed 'search expression' > file.mod
sed 'search expression' file > file.mod
```

8. **sed with substitution:** sed's major use for most of us is in changing the contents of files using the substitution feature. Subsitution uses the following expression:

```
# substitution syntax
sed '[position]s/find/change/flag' file > file.mod
# where
# [position] - optional - normally called address in most documentation
#  s         - indicates substitution command
#  find      - the expression to be changed
#  change    - the expression to be substituted
#  flag      - controls the actions and may be
#              g = repeat on same line
#              N = Nth occurence only on line
#              p = output line only if find was found!
#              (needs -n option to suppress other lines)
#              w ofile = append line to ofile only if find
#                        was found
# if no flag given changes only the first occurrence of
# find on every line is substituted

# examples
# change every occurrence of abc on every line to def
sed 's/abc/def/g' file > file.mod

# change only 2nd occurrence of abc on every line to def
sed 's/abc/def/2' file > file.mod

# creates file changed consisting of only lines in which
# abc was changed to def
sed 's/abc/def/w changed' file
# functionally identical to above
sed -n 's/abc/def/p' file > changed
```

9. **Line deletion:** sed provides for simple line deletion. The following examples illustrate the syntax and a trivial example:

```
# line delete syntax
sed '/find/d' file > file.mod
# where
# find - find regular expression
# d    - delete command

# delete every comment line (starting with #) in file
sed '/^#/d' file > file.mod
```

10. **Delete vs Replace with null:** If you use the delete feature of sed it deletes the entire line on which 'search expression' appears, which may not be the desired outcome. If all you want to do is delete the 'search expression' from the line then use replace with null. The following examples illustrate the difference:

```
# delete (substitute with null) every occurrence of abc in file
sed 's/abc//g' file > file.mod
```

```
# delete every line with abc in file
sed '/abc/d' file > file.mod
```

11. **Escaping:** You need to escape certain characters when using them as literals using the standard \ technique. This removes the width attribute from html pages that many web editors annoyingly place on every line. The " are used as literals in the expression and are escaped by using \:

```
# delete (substitue with null) every occurrence of width="x" in file
# where x may be pure numeric or a percentage
sed 's/width=\"[0-9.%]*\"//g' file.html > file.mod
```

12. **Delimiters:** If you use sed when working with, say, paths which contain / it can be a royal pain to escape them all so you can use any sensible delimiter that minimizes visual confusion for the expresssions. The following example illustrates the principle:

```
# use of / delimiter with a path containing /
# replaces all occurences of /var/www/ with /var/local/www/
sed 's/\/var\/www\//\/var\/local\/www\//g' file > file.mod

# functionally identical but less confusing using : as a 'sensible' delimiter
sed 's:/var/www/:/var/local/www/:g' file > file.mod
```

13. **Positioning with sed:** sed documentation uses, IOHO, the confusing term address for what we call [position]. Positional expressions can optionally be placed before sed commands to position the execution of subsequent expressions/commands. Commands may take 1 or 2 positional expressions which may be line or text based. The following are simple examples:

```
# delete (subsitute with null) every occurrence of abc
# in file only on lines starting with xyz (1 positional expression)
sed '/^xyz/s/abc//g' file > file.mod

# delete (subsitute with null) every occurrence of abc
# only in lines 1 to 50
# 2 positional expression separated by comma
sed '1,50s/abc//g' file > file.mod

# delete (subsitute with null) every occurrence of abc
# except lines 1 - 50
# 2 positional expression separated by comma
sed '1,50!s/abc//g' file > file.mod

# delete (subsitute with null) every occurrence of abc
# between lines containing aaa and xxx
# 2 positional expression separated by comma
sed '/aaa/,/xxx/s/abc//g' file > file.mod

# delete first 50 lines of file
# 2 positional expression separated by comma
sed '1,50d' file > file.mod

# leave first 50 lines of file - delete all others
# 2 positional expression separated by comma
sed '1,50!d' file > file.mod
```

14. **when to use -e:** you can use -e (indicating sed commands) with any search expression but when you have multiple command sequences you must use -e. The

following are functionality identical:

```
# delete (substitute with null) every occurrence of width="x" in file
sed 's/width=\"[0-9.%]*\"//g' file.html > file.mod
sed -e 's/width=\"[0-9.%]*\"//g' file.html > file.mod
```

15. **Strip HTML tags:** Regular expressions take the longest match and therefore when stripping HTML tags may not yield the desired result:

```
# target line
<b>I</b> want you to <i>get</i> lost.

# this command finds the first < and last > on line
sed 's/<.*>//g' file.html
# and yields
lost.

# instead delimit each < with >
sed 's/<[^>]*>//g' file.html
# yields
I want you to get lost.

# finally to allow for multi-line tags you must use
# following (attributed to S.G Ravenhall)
sed -e :aloop -e 's/<[^>]*>//g;/</N;//bloop'
[see explantion below]
```

16. **labels, branching and multiple commands:** sed allows mutiple commands on a single line separated by semi-colons (;) and the definition of labels to allow branching (looping) within commands. The following example illustrates these features:

```
# this sequence strips html tags including multi-line ones
sed -e :aloop -e 's/<[^>]*>//g;/</N;//bloop'

# Explanation:
# -e :aloop consists of :a which creates a label followed by its name
# in this case 'loop' that can be branched to by a later command
# next -e s/<[^>]*>//g; removes tags on a single line and
# the ; terminates this command when the current line is exhausted.
# At this point the line buffer (called the search space) holds the
# current line text with any transformations applied, so <>
# sequences within the line have been removed from the search space.
# However we may have either an < or no < left in the current
# search space which is then processed by the next command which is:
# /</N; which is a positioning command looking for <
# in any remaining part of the search space. If < is found, the N
# adds a NL char to the search space (a delimiter) and tells sed
# to ADD the next line in the file to the search space, control
# then passes to the next command.
# If < was NOT found the search buffer is cleared (output normally)
# and a new line read into the search space as normal. Then control
# passes to the next command, which is:
# //bloop which comprises // = do nothing, b = branch to and loop
# which is the label to which we will branch and was created
# with -e :aloop. This simply restarts the sequence with EITHER just
# the next line of input (no < was left in the search space)
# or with the next line ADDED to the search space (there was a <
# left in the search space but no corresponding >)
# all pretty obvious really!
```

17. **adding line numbers to files:** Sometimes it's incredibly useful to be able to find the line number within a file, say, to match up with error messages for example a parser outputs the message 'error at line 327'. The following adds a line number followed by a single space to each line in file:

```
# add the line number followed by space to every line in file
sed = file|sed 's/\n/ /' > file.lineno
# the first pass (sed = file) creates a line number and
# terminates it with \n (creating a new line)
# the second piped pass (sed 's/\n/ /') substitutes a space
# for \n making a single line
```

Note: We got email asking why the above does not also remove the real end of line (EOL) as well. OK. When any data is read into the buffer for processing the EOL is removed (and appended again only if written to a file). The line number created by the first command is pre-pended (with an EOL to force a new line if written to file) to the processing buffer and the whole lot piped to the second command and thus is the only EOL found.

Problems, comments, suggestions, corrections (including broken links) or something to add? Please take the time from a busy life to 'mail us' (at top of screen), the webmaster (below) or info-support at zytrax. You will have a warm inner glow for the rest of the day.

site by zytrax

web-master at zytrax
Page modified: June 06 2014.