Regular Expressions

Pattern Matching Operators

Match – m//

Syntax:  
m/pattern/

If a match is found for the pattern within a referenced string (default $_), the expression returns true.
(Note: If the delimiters // are used, the preceding m is not required.)

Modifiers:  g, i, m, o, s, x

Substitution – s///

Syntax:  
s/pattern1/pattern2/

If a match is found for pattern1 within a referenced string (default $_), the relevant substring is replaced by the contents of pattern2, and the expression returns true.

Modifiers:  e, g, i, m, o, s, x

Transliteration – tr/// or y///

Syntax:  
tr/pattern1/pattern2/
y/pattern1/pattern2/

If any characters in pattern1 match those within a referenced string (default $_), instances of each are replaced by the corresponding character in pattern2, and the expression returns the number of characters replaced. (Note: If one character occurs several times within pattern1, only the first will be used – for example, tr/abbc/xyz/ is equivalent to tr/abc/xyz/.)

Modifiers:  c, d, s
Appendix A

**Delimiters**
Patterns may be delimited by character pairs <>, (), [], {}, or any other non-word character, e.g.:

```plaintext
s<pattern1><pattern2>
```

and

```plaintext
s#pattern1#pattern2#
```

are both equivalent to

```plaintext
s/pattern1/pattern2/
```

**Binding Operators**

**Binding Operator =~**

Syntax: `$refstring =~ m/pattern/`

Binds a match operator to a variable other than $_. Returns true if a match is found.

**Negation Operator !~**

Syntax: `$refstring !~ m/pattern/`

Binds a match operator to a variable other than $_. Returns true if a match is not found.

**Modifiers**

**Match and Substitution**

The following can be used to modify the behavior of match and substitution operators:

**Cancel Position Reset - /c**

Used only with global matches, that is, as `m//gc`, to prevent the search cursor returning to the start of the string if a match cannot be found. Instead, it remains at the end of the last match found.

**Evaluate Replacement – /e**

Evaluates the second argument of the substitution operator as an expression.

**Global Match – /g**

Finds all the instances in which the pattern matches the string rather than stopping at the first match. Multiple matches will be numbered in the operator's return value.

**Case-Insensitive – /i**

Matches pattern against string while ignoring the case of the characters in either pattern or string.
**Multi-Line Mode – /m**
The string to be matched against is to be regarded as a collection of separate lines, with the result that the metacharacters ^ and $, which would otherwise just match the beginning and end of the entire text, now also match the beginning and end of each line.

**One-Time Pattern Compilation - /o**
If a pattern to match against a string contains variables, these are interpolated to form part of the pattern. Later these variables may change, and the pattern will change with it when next matched against. By adding /o, the pattern will be formed once and will not be recompiled even if the variables within have changed value.

**Single-Line Mode – /s**
The string to be matched against will be regarded as a single line of text, with the result that the metacharacter . will match against the newline character, which it would not do otherwise.

**Free-Form – /x**
Allows the use of whitespace and comments inside a match to expand and explain the expression.

**Transliteration**
The following can be used to modify the behavior of the transliteration operator:

**Complement - /c**
Uses complement of pattern1 – substitutes all characters except those specified in pattern1.

**Delete - /d**
Deletes all the characters found but not replaced.

**Squash - /s**
Multiple replaced characters squashed - only returned once to transliterated string.

**Localized Modifiers**
Syntax:

```
/CaseSensitiveTxt((?i)CaseInsensitiveTxt)CaseSensitiveText/
/CaseInsensitiveTxt((?-i)CaseSensitiveTxt)CaseInsensitiveText/i
```

The following inline modifiers can be placed within a regular expression to enforce or negate relevant matching behavior on limited portions of the expression:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
<th>inline enforce</th>
<th>inline negate</th>
</tr>
</thead>
<tbody>
<tr>
<td>/i</td>
<td>case insensitive</td>
<td>(?i)</td>
<td>(?-i)</td>
</tr>
<tr>
<td>/s</td>
<td>single-line mode</td>
<td>(?s)</td>
<td>(?-s)</td>
</tr>
<tr>
<td>/m</td>
<td>multi-line mode</td>
<td>(?m)</td>
<td>(?-m)</td>
</tr>
<tr>
<td>/x</td>
<td>free-form</td>
<td>(?x)</td>
<td>(?-x)</td>
</tr>
</tbody>
</table>
Metacharacters

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abc]</td>
<td>Any one of a, b, or c.</td>
</tr>
<tr>
<td>[^abc]</td>
<td>Anything other than a, b, and c.</td>
</tr>
<tr>
<td>\d \D</td>
<td>A digit; a non-digit.</td>
</tr>
<tr>
<td>\w \W</td>
<td>A 'word' character; a non-'word' character.</td>
</tr>
<tr>
<td>\s \S</td>
<td>A whitespace character; a non-whitespace character.</td>
</tr>
<tr>
<td>\b</td>
<td>The boundary between a \w character and a \W character.</td>
</tr>
<tr>
<td>-</td>
<td>Any single character (apart from a new line).</td>
</tr>
<tr>
<td>(abc)</td>
<td>The phrase 'abc' as a group.</td>
</tr>
<tr>
<td>?</td>
<td>Preceding character or group may be present 0 or 1 times.</td>
</tr>
<tr>
<td>+</td>
<td>Preceding character or group is present 1 or more times.</td>
</tr>
<tr>
<td>*</td>
<td>Preceding character or group may be present 0 or more times.</td>
</tr>
<tr>
<td>{x,y}</td>
<td>Preceding character or group is present between x and y times.</td>
</tr>
<tr>
<td>{,y}</td>
<td>Preceding character or group is present at most y times.</td>
</tr>
<tr>
<td>{x}</td>
<td>Preceding character or group is present at least x times.</td>
</tr>
<tr>
<td>{x}</td>
<td>Preceding character or group is present x times.</td>
</tr>
</tbody>
</table>

Non-greediness For Quantifiers

Syntax:    (pattern)+?
           (pattern)*?

The metacharacters + and * are greedy by default and will try to match as much as possible of the referenced string (while still achieving a full pattern match). This 'greedy' behavior can be turned off by placing a ? immediately after the respective metacharacter. A non-greedy match finds the minimum number of characters matching the pattern.

Grouping and Alternation

<table>
<thead>
<tr>
<th>For Alternation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
</tr>
</tbody>
</table>

By separating two patterns with |, we can specify that a match on one or the other should be attempted.
() For Grouping And Backreferences ('Capturing')
Syntax: (pattern)
This will group elements in pattern. If those elements are matched, a backreference is made to one of the numeric special variables ($1, $2, $3 etc.)

(?:) For Non-backreferenced Grouping ('Clustering')
Syntax: (?:pattern)
This will group elements in pattern without making backreferences.

Lookahead/behind Assertions

(?!=) For Positive Lookahead
Syntax: pattern1(?!=pattern2)
This lets us look for a match on 'pattern1 followed by pattern2', without backreferencing pattern2.

(?!?) For Negative Lookahead
Syntax: pattern1(?!pattern2)
This lets us look for a match on 'pattern1 not followed by pattern2', without backreferencing pattern2.

(?<=) For Positive Lookbehind
Syntax: pattern1(?<=pattern2)
This lets us look for a match on 'pattern1 preceded by pattern2', without backreferencing pattern2. This only works if pattern2 is of fixed width.

(?<!) For Negative Lookbehind
Syntax: pattern1(?<!pattern2)
This lets us look for a match on 'pattern1 not preceded by pattern2', without backreferencing pattern2. This only works if pattern2 is of fixed width.
Backreference Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\num (num = 1, 2, 3...)</td>
<td>Within a regular expression, \num returns the substring that was matched with the numth grouped pattern in that regexp.</td>
</tr>
<tr>
<td>$num (num = 1, 2, 3...)</td>
<td>Outside a regular expression, $num returns the substring that was matched with the numth grouped pattern in that regexp.</td>
</tr>
<tr>
<td>$+</td>
<td>This returns the substring matched with the last grouped pattern in a regexp.</td>
</tr>
<tr>
<td>$&amp;</td>
<td>This returns the string that matched the whole regexp – this will include portions of the string that matched (?:) groups, which are otherwise not backreferenced.</td>
</tr>
<tr>
<td>$`</td>
<td>This returns everything preceding the matched string in $&amp;.</td>
</tr>
<tr>
<td>$'</td>
<td>This returns everything following the matched string in $&amp;.</td>
</tr>
</tbody>
</table>

Other

(??) For Comments

Syntax: (??comment_text)

This lets us place comments within the body of a regular expression – an alternative to the /x modifier.
Regular Expressions
This work is licensed under the Creative Commons Attribution-NoDerivs-NonCommercial License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd-nc/1.0 or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

The key terms of this license are:

**Attribution:** The licensor permits others to copy, distribute, display, and perform the work. In return, licensees must give the original author credit.

**No Derivative Works:** The licensor permits others to copy, distribute, display and perform only unaltered copies of the work -- not derivative works based on it.

**Noncommercial:** The licensor permits others to copy, distribute, display, and perform the work. In return, licensees may not use the work for commercial purposes -- unless they get the licensor’s permission.