Describing regular expressions

This section includes the following topics:

Precise provides an extensive set of regular expression tools that allow you to efficiently validate strings for assertions as well as to mask different sub-strings for usage in variables. The set of tools is almost identical for both needs.

The descriptions in this section do not cover all regular expressions.

Describing regular expression used in assertions

The regular expressions that are used in assertions are used to verify that an asserted text has a certain "structure" to it. To achieve that, no capture group is needed (see the Groups and Structures table for the definition of a capture group).

The text is matched using the regular expression rules, and the result may be "Matching" or "Not Matching."

Examples:

1. "Hello". The text Hello should be anywhere within the asserted text. Asserting "Hello World" text with this regular expression will succeed.
2. "\AHello\z". The text Hello should be the entire asserted text (between the starting point \A to the ending point \z). Asserting "Hello World" text with this regular expression will fail. The only string that can match it is "Hello" itself.
3. "[0-9]{1,2}\.[0-9]{1,2}\.[0-9]{4}". Text should include a date in the form "xx.xx.xxxx" (where the day and the month may be also one digit). For example, a possible match may be "12.5.2007".

Describing regular expression used in masking

These regular expressions are used in component variables to mask a portion of a text for use as variable value. To achieve that a capture group is needed (see the Groups and Structures table for the definition of a capture group). If several capture groups exist, only the first group's capture is used as a mask. When a capture group exists, all regular expressions within it (inside the regular parenthesis) are selected, and all outside expressions are only used for reference.

Examples:

1. "(Hello)". Captures the text Hello (if exists).
2. "([\w\W]*)". The default expression. Captures the entire text.
3. "(?:[\w\W]+?\s+){8}([\w\W]+?)\s+". Captures the 9th word. The first parenthesis (with the "?:" symbol) is a non-capturing group. The expression inside it is grouped for quantity needs (adding the "{8}" outside), but it will not be captured. The entire expression means. pass 8 words (followed by space characters), capture a word, and finish with space characters (non-captured). The question marks that follow the quantifiers are used to capture the minimal amount of characters that fulfill the quantifier rules.

Describing groups and structure

Groups allow you to capture sub expressions to increase the efficiency of regular expressions using quantifiers for groups, and supply you the method of selecting a sub expression for regular expression masking. The following table shows the list of groups.

Table 1 Groups

Construct	Matches
(X)	The capture group. For regular expression masking, there must be one and only one capture group, matching the selected masked expression. For regular expression assertions the capture group acts the same as the non-capturing group.
(?:X)	A non-capturing group. Any quantifier placed after it refers to the entire group X. For example, "index.(?:a|j)sp" matches index.asp and index.jsp.
XY	X followed by Y.
X|Y	Logical OR (alternation). Either X or Y, The left most successful match wins. Usually used in a group (with parentheses).

Describing characters

Most of the important regular expression language operators are single characters without the escape character. The escape character \ (a single backslash) signals to the regular expression parser that the character following the backslash is not an operator. For example, the parser treats an asterisk (*) as a repeating quantifier and a backslash followed by an asterisk (*) as the Unicode character 002A. The following table shows a list of characters used in regular expressions.

Table 2 Characters

Contruct	Matches
x	The character x.
\\	The backslash character.
\xhh	The character with hexadecimal value 0xhh.
\uhhhh	The character with hexadecimal value 0xhhhh.
\cx	The control character corresponding to x. For example, \cM matches the carriage return character. If x is not in the range of A-Z or a-z, c is assumed to be the literal "c" character.
\t	The tab character ('\u0009'). Equivalent to \cI.
\n	The newline (line feed) character ('\u000A'). Equivalent to \cJ.
\r	The carriage-return character ('\u000D'). Equivalent to \cM.
\f	The form-feed character ('\u000C'). Equivalent to \cL.
\a	The alert (bell) character ('\u0007'). Equivalent to \cG.
\e	The escape character ('\u001B').

Describing character classes

A character class is a set of characters that will find a match if any one of the characters included in the set matches. The following table summarizes character matching syntax.

Table 3 Character classes

Construct	Matches
.	Any character (excluding the newline characters).
[chars]	Any single character included in the specified set of characters. For example, "f[ai]t" matches fat and fit but not fait.
[^chars]	Any single character not in the specified set of characters (negation). For example, "p[^oi]t" matches pat and put but not pot and pit.
[x-y]	Range. Matches any character in the specified range. "[a-z]" matches any lowercase character from a to z, "[A-Z]" matches any uppercase character from A to Z, "[0-9]" matches any integer between 0 and 9, and so on.
\p{name}	Matches any character in the named character class specified by {name}. Supported names are Unicode groups and block ranges. For example, Ll, Nd, Z, IsGreek, and so on.
\P{name}	Matches text not included in groups and block ranges specified in {name}.
\d	A decimal digit: [0-9].
\D	A non-digit: [^0-9].
\s	A whitespace character: [\t\n\x0B\f\r].
\S	A non-whitespace character: [^\s].
\w	A word character: [a-zA-Z_0-9].
\W	A non-word character: [^\w].

Describing atomic zero-width matches

The meta characters described in the following table do not represent characters. They simply cause a match to succeed or fail depending on the current position in the string. For example, ^ specifies that the current position is at the beginning of a line. Thus, the regular expression ^FTP returns only those occurrences of the character string "FTP" that occur at the beginning of a line.

Table 4 Meta characters

Construct	Matches
^	The beginning of a line (any line in a multi-line input).
$	The end of a line (before \n at the end of the line).
\b	A word boundary - that is, at the first or last characters in words separated by any non-alphanumeric characters.
\B	Not a \b boundary.
\A	The beginning of the input.
\Z	The end of the input (before \n at the end of the input, if exists).
\z	The end of the input.

Describing quantifiers

Quantifiers add optional quantity data to a regular expression. A quantifier expression applies to the character, group, or character class that immediately precedes it. When more than one match exists, it returns the maximum number of repetitions. The following table describes the quantifiers. The minimum number of repetitions is described in Lazy Quantifiers.

Table 5 Quantifiers

Construct	Matches
X?	X, once or not at all. For example, "potatoe?" matches “potato” and “potatoe”. When this quantifier follows another quantifier, the result is a lazy quantifier. For more information, see the Lazy Quantifiers table.
X*	X, zero or more times. For example, "Gr*" matches G, Gr, Grr, Grrr, and so on (in this example the * defines that the “r” is either there or not).
X+	X, one or more times. For example, "Gr+" matches Gr, Grr, Grrr, and so on.
X{n}	X, exactly n times. For example, "(?:ab){2}" matches "abab".
X{n,}	X, at least n times. For example, "[1-4]{2,}" matches any 2+ digit number with the combination of numbers 1 through 4.
X{n,m}	X, at least n but not more than m times. For example, "mai{0,1}n" matches main and man.

You cannot use quantifiers with atomic zero-width matches.

Describing lazy quantifiers

Lazy quantifiers behave the same as quantifiers, except that instead of searching the maximum number of repetitions, it searches for the minimum number of repetitions. The following table describes the lazy quantifiers.

Table 6 Lazy quantifiers

Construct	Matches
X??	X, not at all if possible, and once. Lazy X?.
X*?	X, zero or more times (with as few repeats as possible). Lazy X*.
X+?	X, one or more times (with as few repeats as possible). Lazy X+.
X{n}?	Equivalent to {n}.
X{n,}?	X, at least n times with as few repetitions as possible. Lazy X{n,}.
X{n,m}?	X, at least n but not more than m times, with as few repetitions as possible. Lazy X{n,m}.

For additional information about regular expressions, see the website http://en.wikipedia.org/wiki/Regular_expression.