Regular Expression (Regexe)

Regular Expression, or *regexe* in short, is extremely and amazingly powerful in searching and manipulating text strings. One line of regexe can easily replace several dozen lines of programming codes. Regexe is supported in almost all the scripting languages (such as Perl, Python, PHP, and JavaScript) as well as general purpose programming languages such as Java, and even word processor such as Word for searching texts. Getting started with regexe may not be easy due to its geeky syntax, but it is certainly worth the investment of your time.

Regexe By Examples

Example 1: Numeric String `/^[0-9]+$/`

1. A regexe is typically delimited by a pair of forward slash, in the form of `/.../`.
2. The leading `^` and the trailing `$` are known as position anchors, which match the beginning and ending of the input string, respectively. As a result, the entire input string shall be matched, instead of a portion of the input string. Without these position anchors, the regexe can match any part of the input string, i.e., with leading and trailing sub-string unmatched.
3. The `[0-9]` matches any character between 0 and 9, i.e., all digits.
4. The `+` indicate 1 or more occurrences of the previous sub-expression. In this case, `[0-9]+` matches one or more digits.
5. This regexe matches any non-empty numeric string (of digits 0 to 9), e.g., "`0`", "`12345`". It does not match with "" (empty string), "`abc`", "`a123`", etc.

Example 2: Integer Literal `/[1-9][0-9]*|0/`

[TODO]

Example 3: An Identifier (or Name) `/[a-zA-Z_][a-zA-Z_]*/`

[TODO]

Example 4: An Image Filename `\^\S+\.(gif|png|jpg|jpeg)$\i`

1. The two forward-slashes `/.../` contains a regexe.
2. The leading `^` and trailing `$` match the beginning and the ending of the input string, respectively. That is, the entire input string shall match with this regexe, instead of a part of the input string.
3. `\S+` matches one or more non-whitespaces.
4. `\.` matches the `.` character. We need to use `\.` to represent `.` as `.` has special meaning in regexe. The `\` is known as the escape code, which restore the original literal meaning of the following character.
5. `(gif|png|jpg|jpeg)` matches either "`gif`", "`png`", "`jpg`" or "`jpeg`". The `|` means "or".
6. The modifier `i` after the regexe specifies case-insensitive matching. That is, it accepts "`test.GIF`" and "`test.Gif`".

Example 5: Email Address `/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/`

1. The two forward-slashes `/.../` contains a regexe.
2. The leading `^` and trailing `$` match the beginning and the ending of the input string, respectively. That is, the entire input string shall match with this regexe, instead of a part of the input string.
3. `\w+` matches 1 or more word characters (a-z, A-Z, 0-9 and underscore).
4. `[\.-]` matches character `.` or `-`. We need to use `\.` to represent `.` as `.` has special meaning in regexe. The `\` is known as the escape code, which restore the original literal meaning of the following character.
5. `[\.-]?` matches 0 or 1 occurrence of `[\.-]`.
6. Again, `\w+` matches 1 or more word characters.
7. `([\.-]?\w+)*` matches 0 or more occurrences of `[\.-]?\w+`.
8. The sub-expression `\w+([\.-]?\w+)*` is used to match the username in the email, before the `@` sign. It begins with at least one word character (a-z, A-Z, 0-9 and underscore), followed by more word characters or `.` or `-`. However, a `.` or `-` must follow by a word character (a-z, A-Z, 0-9 and underscore). That is, the string cannot contain "`..`", "`--`", "`.-`" or "`-.`". Example of valid string are "`a.1-2-3`".
9. The `@` matches itself.
10. Again, the sub-expression `\w+([\.-]?\w+)*` is used to match the email domain name, with the same pattern as the username described above.
11. The sub-expression `\.\w{2,3}` matches a . followed by two or three word characters, e.g., "`.com`", "`.edu`", "`.us`", "`.uk`", "`.co`".
12. `(\.\w{2,3})+` specifies that the above sub-expression shall occur one or more times, e.g., "`.com`", "`.co.uk`", "`.edu.sg`" etc.

Exercise: Interpret this regexe, which provide another representation of email address: `/^[\w\-\.\+]+\@[a-zA-Z0-9\.\-]+\.[a-zA-z0-9]{2,4}$/`.

Example 6: Swapping the First two words using Back-References `/^(\S+)\s+(\S+)$/`

1. The two forward-slashes `/.../` contain a regexe.
2. The `^` and `$` match the beginning and ending of the input string, respectively
3. `\S+` (uppercase `S`) matches one or more non-whitespaces; \s+ (lowercase `s`) matches one or more whitespaces (blank, tab and newline). In regexe, the uppercase metacharacter denotes the *inverse* of the lowercase counterpart. For example, `\s` for whitespace and `\S` for non-whitespace.
4. The parentheses in `(\S+)`, called *parenthesized back-reference*, is used to extract the matched sub-string from the input string. In this regexe, there are two `(\S+)`, match the first two words, separated by one or more whitespaces `\s+`. The two matched words are extracted from the input string and kept in special variables `$1` and `$2` respectively.
5. To reverse the first two words, you can access the special variables, and print "`$2 $1`" (via a programming language).

Example 7: HTTP Address `/^http:\/\/\S+(\/\S+)*(\/)?$/`

[TODO] more

Regexe Syntax

A *Regular Expression* (or *Regexe*) is a *pattern* (or *filter*) that accepts a set of strings that *matches* the pattern.

A regexe is typically delimited by a pair of forward slashes `/.../`. It consists of a sequence of characters, meta-characters (such as `.`, `\d`, `\s`, `\S`) and operators (such as `+`, `*`, `?`, `|`, `^`)

For examples,

Regexe	Matches
`/^[0-9]+$/`	A numeric string with at least one digits
`/^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,3})+$/`	A valid email address

Sub-Expressions

A regexe is constructed by combining many smaller *sub-expressions*.

Matching a Character

The fundamental building blocks are patterns that match *a single character*. Most characters, including all letters and digits, match themselves. For example, the regexe `/Friday/` matches the string "`Friday`".

To match a special character (such as `+`, `*`, `/`, `\`, which are regexe's operators), you need to precede the character with a backslash, known as *escaping* (i.e., `\+`, `\*`, `\/` and `\\`). For example, the regexe `/a\*b\+/` matches the string "`a*b+`".

OR ('|') Operator

You can provide *alternatives* using the "OR" operator, denoted by a vertical bar `'|'`.

For example: The regexe `/four|for|floor|4/` matches strings "four", "for", "floor" or "4"

Bracket `[ ]` and Range `[ - ]` Expressions

A *bracket expression* is *a list of characters* enclosed by `[ ]`, also called *character class*. It matches any single character in the list. However, if the first character of the list is the caret (`^`), then it matches any single character NOT in the list. For example, the regexes `[02468]` matches a single digit 0, 2, 4, 6, or 8; the regexes `[^02468]` matches any single character other than 0, 2, 4, 6, or 8.

Instead of listing all characters, you could use a *range expression* inside the bracket. A range expression consists of 2 characters separated by a hyphen (`-`). It matches any single character that sorts between the two characters, inclusive. For example, `[a-d]` is the same as `[abcd]`. You could include a caret (`^`) in front of the range to invert the matching. For example, `[^a-d]` is equivalent to `[^abcd]`.

Special characters: To include a `]`, place it first in the list. To include a `^`, place it anywhere but first. Finally, to include a `-` place it last. Most metacharacters (such as `+`, `*`) lose their special meaning inside bracketed lists.

Occurrence Indicators (aka Repetition Operators): `+`, `*`, `?`, `{}`

A regexe sub-expression may be followed by an *occurrence indicator* (aka *repetition operator*):

- `?`: The preceding item is optional and matched at most once (i.e., occurs 0 or 1 times).
- `*`: The preceding item will be matched zero or more times.
- `+`: The preceding item will be matched one or more times.
- `{m}`: The preceding item is matched exactly m times.
- `{m,}`: The preceding item is matched m or more times.
- `{m,n}`: The preceding item is matched at least m times, but not more than n times.

For example: The regexe `/xy{2,4}/` (`'x'` followed by 2 to 4 `'y'`) matches "`xyy`", "`xyyy`" and "`xyyyy`".

Greediness

`*`, `+`, `?`, `{ }` repetition operators are *greedy operators*, and by default grasp as many characters as possible for a match. For example, the regexe `/xy{2,4}/` try to match for "xyyyy", then "xyyy", and then "xyy".

In some languages like Perl, you can put an extra `?` after the repetition operator (`*`, `+`, `?`, or `{}`) to curb its greediness (i.e., stop at the shortest match) in the form of `*?`, `+?`, `??`, `{}?`. For example,

```
input = "The <code>first</code> and <code>second</code> instances"
/<code>.*</code>/g matches "<code>first</code> and <code>second</code>".
But
/<code>.*?</code>/g matches "<code>first</code>", "<code>second</code>".
```

Metacharacters `.`, `\w`, `\W`, `\d`, `\D`, `\s`, `\S`, `\xnn`, `\onn`

A *metacharacter* is a symbol with a special meaning inside a regexe.

- The metacharacter dot (`.`) matches any single character except newline `\n` (same as `[^\n]`). For example, `/.../` matches any 3 characters (including alphabets, numbers, white-spaces, but except newline); `/the../` matches "`there`", "`these`", "`the  `" and etc.
- `\w` (word character) matches any single letter, number or underscore (same as `[a-zA-Z0-9_]`). The uppercase counterpart `\W` (non-word-character) matches any single character that doesn't match by `\w` (same as `[^a-zA-Z0-9_]`). In regexe, the uppercase metacharacter is always the inverse of the lowercase counterpart.
- `\d` (digit) matches any single digit (same as `[0-9]`). The uppercase counterpart `\D` (non-digit) matches any single character that is not a digit (same as `[^0-9]`).
- `\s` (space) matches any single whitespace (same as `[\t\n\r\f ]`). The uppercase counterpart `\S` (non-space) matches any single character that doesn't match by `\s` (same as `[^\t\n\r\f ]`).
- `\xnn` matches hexadecimal number *nn*; and `\onn` matches octal number *nn*.

Examples:

```
/\s\s/      # Matches two whitespaces
/\S\S\s/    # Two non-whitespaces followed by a whitespace
/\s+/       # one or more whitespaces
/\S+\s\S+/  # two words (non-whitespaces) separated by a whitespace
```

Positional Metacharacters (aka Position Anchors) `^`, `$`, `\b`, `\B`, `\<`, `\>`, `\A`, `\Z`

Positional anchors do NOT match actual character but matches *position* in a string, such as begin-of-line, end-of-line, begin-of-word, and end-of-word.

- The metacharacter caret (`^`) matches the beginning of the line (or input string); and the metacharacter dollar (`$`) matches the end of line, excluding newline (or end of input string). These two are the most commonly used position anchors.
- The metacharacter `\b` matches the the edge of a word (i.e., word boundary after a whitespace); and `\B` matches a string provided it's not at the edge of a word. For example, `/\bcat\b/` matches the word "`cat`" (excluding the leading and trailing whitespaces).
- The metacharacters `\<` and `\>` match the empty string at the beginning and the end of a word respectively (compared with `\b`, which can match both the beginning and the end of a word).
- `\A` matches the beginning of the line, just like `^`. `\Z` matches the end of the line, just like `$`. However, `\A` and `\Z` match only the actual beginning and ending, and ignore the embedded newlines within the string.

You can use positional anchors liberally to increase the speed of matching. For examples:

```
/ing$/              # ending with 'ing'
/^testing 123$/     # Matches only one pattern. Should use equality comparison instead.
```

Parenthesized Back-References & Matched Variables `$1,... , $9`

Parentheses `( )` serve two purposes in regexes.

1. Firstly, parentheses `( )` can be used to group sub-expressions for overriding the precedence or applying a repetition operator. For example, `/(a|e|i|o|u){3,5}/` is the same as `/a{3,5}|e{3,5}|i{3,5}|o{3,5}|u{3,5}/`.
2. Secondly, parentheses are used to provide the so called *back-references*. A back-reference contains the *matched sub-string*. For examples, the regexe `/(\S+)/` creates one back-reference `(\S+)`, which contains the first word (consecutive non-spaces) of the input string; the regexe `/(\S+)\s+(\S+)/` creates two back-references: `(\S+)` and another `(\S+)`, containing the first two words, separated by one or more spaces `\s+`.

The back-references are stored in special variables `$1`, `$2`, …, `$9`, where `$1` contains the substring matched the first pair of parentheses, and so on. For example, `/(\S+)\s+(\S+)/` creates two back-references which matched with the first two words. The matched words are stored in `$1` and `$2`, respectively.

Back-references are important if you wish to manipulate the string.

For example, the following Perl expression swap the first and second words separate by a space:

```
s/(\S+) (\S+)/$2 $1/;    # Swap the first and second words separated by a single space
```

Modifier

You can attach *modifiers* after a regexe, in the form of `/.../`*`modifiers`*, to control the matching behavior. For example,

- `i`: case insensitive matching. The default is case-sensitive.
- `g`: global matching, i.e., search for all the matches. The default searches only the first match.

Regexe in Programming Languages

JavaScript: [Link]

Java: [Link]

Perl: [Link]

REFERENCES & RESOURCES

- TODO

Last modified: October, 2011