***Regular Expressions***

**Introduction**

Regular expression are nothing but sequence of alphanumerical which specify certain pattern ,which mainly use in pattern matching(string matching).it works like sql "like" operations. The phrase regular expression is mainly used to mean to be specific, standard textual syntax for representing patterns that matching text need to conform to. The concept arose in the 1950s, when the American mathematician Stephen Kleene formalized the description of a regular language, and came into common use with the Unix text processing utilities ed, an editor, and grep, a filter. Many [programming languages](https://en.wikipedia.org/wiki/Programming_language) provide regular expression capabilities, some built-in (for example [Perl](https://en.wikipedia.org/wiki/Perl_language_structure#Regular_expressions), [JavaScript](https://en.wikipedia.org/wiki/JavaScript), [Ruby](https://en.wikipedia.org/wiki/Ruby_%28programming_language%29), [AWK](https://en.wikipedia.org/wiki/AWK), and [Tcl](https://en.wikipedia.org/wiki/Tcl)) , other languages via a [standard library](https://en.wikipedia.org/wiki/Standard_library) (for example [.NET languages](https://en.wikipedia.org/wiki/.NET_Framework), [Java](https://en.wikipedia.org/wiki/Java_%28programming_language%29), [Python](https://en.wikipedia.org/wiki/Python_%28programming_language%29), [POSIX C](https://en.wikipedia.org/wiki/C_POSIX_library#regex.h), and [C++](https://en.wikipedia.org/wiki/C%2B%2B) since [C++11](https://en.wikipedia.org/wiki/C%2B%2B11#regex)). Most other languages offer regular expressions via a library

Each character in a regular expression (that is, each character in the string describing its pattern) is said to be as Meta character, or also called as a regular character. Together, they are used for identifying textual matter of a given pattern, or process a number of instances of it. Pattern-matches can vary from a precise equality to a very general similarity (controlled by the meta characters). The meta character syntax is designed specifically to represent prescribed targets in a concise and flexible way to direct the automation of text processing of a variety of input data, in a form easy to type using standard ASCII keyboard.

**Basic Concepts**

Regular Expression also called as pattern it is an expression which specify set of strings to be validate for a specific purpose. One of the way to specify finite set of string is list its all members(elements).for eg.a set contain 2 strings "Color", "Colour", then we can specify the pattern as "Col(ou?)r", here we match pattern matches any of this two strings .if there exists at least one regular expression that matches a particular set then there exists an infinite number of other regular expression that also match it the specification is not unique.

* **Boolean "or"**

A [vertical bar](https://en.wikipedia.org/wiki/Vertical_bar) **(“|”)** separates alternatives. For example, colour|color can match "colour" or "color".

* **Grouping**

[Parentheses](https://en.wikipedia.org/wiki/Bracket)**”()”** are used to define the precedence and scope of the [operators](https://en.wikipedia.org/wiki/Operator_%28programming%29) . For example, color|colour and gr(o|ou)y are equivalent patterns which both describe the set of "color" or "colour".

* **Quantification**

A [quantifier](https://en.wikipedia.org/wiki/Quantifier_%28linguistics%29) after string specifies how often preceding token(element) is allowed to occur. Question mark(“**?**”),asterisk(“\*”)and plus sign(“**+**”) are the most common quantifiers.

|  |  |
| --- | --- |
| **Meta Characters** | **Description** |
| **?** | The question mark indicates *zero or one* occurrences of the preceding element. For example, colou?r matches both "color" and "colour". |
| **??** | Repeat 0 times or 1 time matching 0 times if possible. |
| **\*** | The asterisk indicates *zero or more* occurrences of the preceding element. For example, ab\*c matches "ac", "abc", "abbc", "abbbc", and so on. |
| **\*?** | Repeat 0 or more times matching as few times as possible. |
| **+** | The plus sign indicates *one or more* occurrences of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac". |
| **+?** | Repeat 1 or more times matching as few times as possible. |
| **{n}** | The preceding item is matched exactly *n* times. |
| **{min,}** | The preceding item is matched *min* or more times. |
| **{min,}?** | Repeat at least *n* times, matching as few times as possible. |
| **{min,max}** | The preceding item is matched at least *min* times, but not more than *max* times. |
| **{min,max}?** | Repeat at least *n* times, but no more than *m* times while matching as few as possible. |

### Delimiters

When using any regular expression in a programming language, they may be represented as a usual string literal, so mostly they are quoted. Here the regular expression so is entered as “so”. However, they are often written with slashes as [delimiters](https://en.wikipedia.org/wiki/Delimiter), as in /so/ for the regular expression. / was originated from editor command for searching /so/ could be use to combine commands like a/s/p as in g/re/p as in grep ("global regexp print").Another convention used for search and replace is ",".The string patterns can be concatenated by a comma to specify range of lines as /rel1/,/rel2/. Consider the example: command C, /, # will replace a / with an #, using commas as delimiters.

### Character classes

### Character class matches single characters. Following table shows character classes which are used in regular expressions

|  |  |
| --- | --- |
| **Element** | **Description** |
| **.** | Matches any character but not the line feed. Includes the line feed in single-line mode. Matches a or 1 or almost everything. |
| **[string]** | Matches any character that is contained within the brackets [ ], in no particular order.suppose we insert a “-“ in between characters of them for eg.[mno-r] will match string,”m”,”o”,”p”,”q”,”r”.it can be also written as [m-o-r] it does return same |
| **[^string]** | The opposite of [ ]. Matches all characters of the string which does not contained within the brackets.eg.[^xyz]will matches anything except x, y, or z . |
| **[a-z]** | Character range: Matches any single character in the range from first (a) to last (z).  [a-z] matches a, m, or z |
| **\w** | Matches an alpha-numeric character (a-z, A-Z, 0-9, and underscore).  \w matches a or b |
| **\W** | The opposite of \w. Matches any non-alphanumeric character.  \W matches - but does not match a |
| **\d** | Matches a decimal character (0-9).  \d matches 1 or 2 |
| **\D** | The opposite of \d. Matches any non-decimal character.  \D matches a or b |
| **\s** | Matches a character of whitespace (space, tab, carriage return, line feed).  a\sb matches a b |
| **\S** | The opposite of \s. Matches any non-whitespace character.  a\Sb matches a-b |
| **\r** | Matches a carriage return.  a\rb matches a b |
| **\n** | Matches a new line (line feed).  a\nb matches a b |
| **\f** | Matches a form feed. |
| **\t** | Matches a tab.  a\tb matches a    b |
| **\v** | Matches a vertical tab. |
| **\a** | Matches a bell character. |
| **\b** | In a character class, matches a backspace. |
| **\e** | Matches an escape. |
| **\040** | Uses octal representation to specify a character (octal consists of up to three digits). |
| **\x20** | Uses hexadecimal representation to specify a character (hex consists of exactly two digits). |
| **\c0003** | Matches the specified 4-digit ASCII control character. |
| **\u0020** | Matches a Unicode character by using hexadecimal representation (exactly four digits). |
| **\p{name}** | Matches any single character in the Unicode general category or named block specified by name. |
| **\P{name}** | Matches any single character that is not in the Unicode general category or named block specified by name. |

### Assertions

### The following table shows expressions that are used to specify location to search for a match, but do not match anything by them.

|  |  |
| --- | --- |
| **Elements** | **Description** |
| **^** | The match must start at the beginning of the string (or beginning of the line in multiline mode).  ^cat matches **cat** but does not match **bobcat** |
| **$** | The match must occur at the end of the string or before \n at the end of the string (or end of the line in multiline mode).  dog$ matches **dog** but does not match **dogfight** |
| **\A** | The match must occur at the start of the string. |
| **\Z** | The match must occur at the end of the string or before \n at the end of the string. |
| **\z** | The match must occur at the end of the string. |
| **\G** | The match must occur at the point where the previous match ended. |
| **\b** | Asserts a boundary between word and non-word characters.  grape\b matches **grape, cherry** but does not match **grapefruit** |
| **\B** | The opposite of \b. Asserts a location that is not a boundary between word and non-word characters.  grape\B matches **grapefruit** but does not match **grape, cherry** |
| **(?=pattern)** | Asserts that the specified *pattern* exists immediately after this location. Known as a positive lookahead.  too many(?= secrets) matches **too many secrets** but does not match **too many** |
| **(?!pattern)** | Asserts that the specified *pattern* does not exist immediately after this location. Known as a negative look ahead.  too many(?! secrets) matches **too many** but does not match **too many secrets** |
| **(?<=pattern)** | Asserts that the specified *pattern* exists immediately before this location. Known as a positive look behind.  (?<=too )many secrets matches **too many secrets** but does not match **many secrets** |
| **(?<!pattern)** | Asserts that the specified *pattern* does not exist immediately before this location. Known as a negative look behind.  (?<!too )many secrets matches **many secrets** but does not match **too many secrets** |

### Grouping

### Following expressions allow pattern matching in a group

|  |  |
| --- | --- |
| **Expression** | **Description** |
| **(pattern)** | Captures the specified pattern as a group. Each group is numbered automatically starting from 1. Group 0 is actually not a group at all but refers to the text matched by the entire regular expression. |
| **(?<name>pattern)** | Captures the specified pattern into the specified group name. The string used for the name must not contain any punctuation and cannot begin with a number. |
| **(?<name1-name2>pattern)** | Defines a balancing group definition. |
| **(?:pattern)** | Does not capture the substring matched by this pattern. Known as a non capturing group. |
| **(?imnsx-imnsx:pattern)** | Applies or disables the specified options within sub expression. |
| **(?>pattern)** | Non backtracking (or "greedy") sub expression. |

### Back references

### A back reference allows a previously matched sub expression to be identified subsequently in the same regular expression.

|  |  |  |
| --- | --- | --- |
| **Expression** | **Description** | |
| **\number** | Back reference. Matches the value of a numbered sub expression. | |
| **\k<name>** | Named back reference. Matches the value of a named expression. | |
| Substitutions |  |

### Substitutions are allowed only within replacement patterns. Following table shows substitution expressions.

|  |  |
| --- | --- |
| **Expression** | **Description** |
| **$number** | Substitutes the last substring matched by the specified group number. The numbering scheme for groups starts at 1 (0 represents the entire match). |
| **${name}** | Substitutes the last substring matched by a named group. |
| **$&** | Substitutes a copy of the entire match itself. |
| **$`** | Substitutes all the text of the input string before the match. |
| **$'** | Substitutes all the text of the input string after the match. |
| **$+** | Substitutes the last group captured. |
| **$\_** | Substitutes the entire input string. |

### Alteration

### Alteration expressions allows either or expression. Following table shows alteration expressions

|  |  |
| --- | --- |
| **Expression** | **Description** |
| **|** | Acts as a logical OR. When between two characters or groups, matches one or the other. |
| **(?(pattern)yes|no)** | Matches the first pattern in the OR statement (yes) if the specified *pattern* is found at this point. Otherwise, matches the second pattern in the OR statement (no). |
| **(?(<name>)yes|no)** | Matches the first pattern in the OR statement (yes) if the specified named group is found at this point. Otherwise, matches the second pattern in the OR statement (no). |

### Comments

### Comments expression allows comments to be inserted in our regular expression. Following table shows comment expressions.

|  |  |
| --- | --- |
| **(?#comment)** | Everything from the pound sign (#) to the end parenthesis is a comment and will be ignored. |
| **#comment** | X-mode comment. The comment starts at an un escaped # and continues to the end of the line. |

### Example

### we can combine any type of expression with other and make a regular expression for our pattern matching and searching.

#### \*How to validate data with 8 digit fix numeric format like 91230456, 01237648 etc?

### ^[0-9]{8}$

#### \*How to validate numeric data with minimum length of 3 and maximum of 7, ex -123, 1274667, 87654?

### ^[0-9]{3,7}$

### \*Validates a name. Allows up to 50 uppercase and lowercase characters and a few special characters that are common to some names.

### ^[a-zA-Z''-'\s]{1,40}$

### Likewise following are the most commonly used validation expressions

### Visa: ^4[0-9]{12}(?:[0-9]{3})?$ All Visa card numbers start with a 4. New cards have 16 digits. Old cards have 13.

### Email: \w+([-+.']\w+)\*@\w+([-.]\w+)\*\.\w+([-.]\w+)\*

### Mobile Number: [0-9]{10}

### URL: http(s)?://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]\*)?

### Pin Code: \d{6}