In theoretical computer science and formal language theory, a regular expression (abbreviated regex or regexp) is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations. 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 (global regular expression print), a filter.

Each character in a regular expression is either understood to be a metacharacter with its special meaning, or a regular character with its literal meaning. Together, they can be used to identify textual material of a given pattern, or process a number of instances of it that can vary from a precise equality to a very general similarity of the pattern. The pattern sequence itself is an expression that is a statement in a language designed specifically to represent prescribed targets in the most concise and flexible way to direct the automation of text files, specific textual forms, or of random input strings.

A very simple use of a regular expression would be to locate the same word spelled two different ways in a text editor, general text files, specific textual forms, or of random input strings.
History [edit]

Regular expressions originated in 1956, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called regular sets.[1] These arose in theoretical computer science, in the subfields of automata theory (models of computation) and the description and classification of formal languages. Other early implementations of pattern matching include the SNOBOL language, which did not use regular expressions, but instead its own syntax.

Regular expressions entered popular use from 1968 in two uses: pattern matching in a text editor[2] and lexical analysis in a compiler.[3] Among the first appearances of regular expressions in program form was when Ken Thompson built Kleene's notation into the editor QED as a means to match patterns in text files.[2][4][5][6] For speed, Thompson implemented regular expression matching by just-in-time compilation (JIT) to IBM 7094 code on the Compatible Time-Sharing System, an important early example of JIT compilation.[7] He later added this capability to the Unix editor ed, which eventually led to the popular search tool grep's use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: g/ze/p meaning "Global search for Regular Expression and Print matching lines").[8] Around the same time when Thompson developed QED, a group of researchers including Douglas T. Ross implemented a tool based on regular expressions that is used for lexical analysis in compiler design.[3]

Many variations of these original forms of regular expressions were used in Unix[6] programs at Bell Labs in the 1970s, including vi, lex, sed, AWK, and expr, and in other programs such as Emacs. Regular expressions were subsequently adopted by a wide range of programs, with these early forms standardized in the POSIX.2 standard in 1992.

In the 1980s more complicated regular expressions arose in Perl, which originally derived from a regex library written by Henry Spencer (1986), who later wrote an implementation of Advanced Regular Expressions for Tcl.[9] The Tc library is a hybrid NFA/DFA implementation with improved performance characteristics, earning praise from Jeffrey Friedl who said, "...it really seems quite wonderful...".[10] Software projects that have adopted Spencer's Tcl regular expression implementation include PostgreSQL.[11] Perl later expanded on Spencer's original library to add many new features,[12] but has not yet caught up with Spencer's Advanced Regular Expressions implementation in terms of performance or Unicode handling.[13][14] Part of the effort in the design of Perl 6 is to improve Perl's regular expression integration, and to increase their scope and capabilities to allow the definition of parsing expression grammars.[15] The result is a mini-language called Perl 6 rules, which are used to define Perl 6 grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regular expressions, but also allow BNF-style definition of a recursive descent parser via sub-rules.

The use of regular expressions in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s when industry standards like ISO SGML (precursored by ANSI "GCA 101-1983") consolidated. The kernel of the structure specification language standards consists of regular expressions. Its use is evident in the DTD element group syntax.

Starting in 1997, Philip Hazel developed PCRE (Perl Compatible Regular Expressions), which attempts to closely mimic Perl's regular expression functionality and is used by many modern tools including PHP and Apache HTTP Server.

Today regular expressions are widely supported in programming languages, text processing programs (particular
Regular expressions, advanced text editors, and some other programs. Regular expression support is part of the standard library of many programming languages, including Java and Python, and is built into the syntax of others, including Perl and ECMAScript. Implementations of regular expression functionality is often called a regular expression engine, and a number of libraries are available for reuse.

Basic concepts  [edit]

A regular expression, often called a pattern, is an expression used to specify a set of strings required for a particular purpose. A simple way to specify a finite set of strings is to list its elements or members. However, there are often more concise ways to specify the desired set of strings. For example, the set containing the three strings "Handel", "Händel", and "Haendel" can be specified by the pattern H(ä|ae?)ndel; we say that this pattern matches each of the three strings. In most formalisms, if there exists at least one regex that matches a particular set then there exists an infinite number of other regex that also match it—the specification is not unique. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"

A vertical bar separates alternatives. For example, gray|grey can match "gray" or "grey".

Grouping

Parentheses are used to define the scope and precedence of the operators (among other uses). For example, gray|grey and gr(ä|e)y are equivalent patterns which both describe the set of "gray" or "grey".

Quantification

A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark ?, the asterisk * (derived from the Kleene star), and the plus sign + (Kleene plus).

?  The question mark indicates there is zero or one of the preceding element. For example, colou?r matches both "color" and "colour".

*  The asterisk indicates there is zero or more of the preceding element. For example, ab*c matches "ac", "abc", "abbc", "abbbc", and so on.

+  The plus sign indicates there is one or more of the preceding element. For example, ab+c matches "abc", "abbc", "abbbc", and so on, but not "ac".

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷. For example, H(ä?|ä)ndel and H(a|ae|ä)ndel are both valid patterns which match the same strings as the earlier example, H(ä|ae?)ndel.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the Syntax section.

Formal language theory  [edit]

Regular expressions describe regular languages in formal language theory. They have the same expressive power as regular grammars.

Formal definition  [edit]

Regular expressions consist of constants and operator symbols that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory.[16][17] Given a finite alphabet Σ, the following constants are defined as regular expressions:

- (empty set) ∅ denoting the set ∅.
- (empty string) ε denoting the set containing only the "empty" string, which has no characters at all.
- (literal character) a in Σ denoting the set containing only the character a.

Given regular expressions R and S, the following operations over them are defined to produce regular expressions:

- (concatenation) RS denotes the set of strings that can be obtained by concatenating a string in R and a string in S. For example "ab", "c")|("d", "ef") = ("ab", "b"ab", "cd", "c"ef").
- (alternation) R | S denotes the set union of sets described by R and S. For example, if R describes ("ab", "c") and S describes ("ab", "d", "ef"), expression R | S describes ("ab", "c", "d", "ef").
- (Kleene star) R* denotes the smallest superset of set described by R that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from set described by R. For example, ("0","1")* is the set of all finite binary strings (including the empty string), and ("ab", "c")* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ...}.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then alternation. If there is no ambiguity then parentheses may be omitted. For example, (ab)c can be written as abc, and a(b(c*)) can be written as abc*. Many textbooks use the symbols U, +, or V for alternation instead of the
Depending on the regexp processor there are about fourteen metacharacters, characters that may or may not have their literal character meaning, depending on context, or whether they are "escaped", i.e. preceded by an escape sequence, in this case, the backslash \. Modern and POSIX extended regular expressions use metacharacters more often than their literal meaning, so to avoid "backslash-osis" it makes sense to have a metacharacter escape to a
Delimiters [edit]

When entering a regular expression in a programming language, they may be represented as a usual string literal, hence usually quoted; this is common in C, Java, and Python, for instance, where the regular expression `re` is entered as "re". However, they are often written with slashes as delimiters, as in `/re/` for the regular expression `re`. This originates in ed, where `/` is the editor command for searching, and an expression `/re/` can be used to specify a range of lines (matching the pattern), which can be combined with other commands on either side, most famously `g/re/p` as in `grep` ("global regex print"). A similar convention is used in `sed`, where search and replace is given by `s/regexp/replacement/` and patterns can be joined with a comma to specify a range of lines as in `/re1,,re2/`. This notation is particularly well-known due to its use in `Perl`, where it forms part of the syntax distinct from normal string literals. In some cases, such as sed and Perl, alternative delimiters can be used to avoid collision with contents, and to avoid having to escape occurrences of the delimiter character in the contents. For example, in `sed` the `s/regexp/replacement/` command will replace a string with `replacement`, and to avoid having to escape occurrences of the delimiter character in the contents. For example, in `sed` the `s/_/x/` command will replace a `_` with an `x`, using commas as delimiters.

Standards [edit]

The [IEEE POSIX] standard has three sets of compliance: BRE,[21] ERE, and SRE for Basic, Extended, and Simple Regular Expressions. SRE is [deprecated][22] in favor of BRE, as both provide backward compatibility. The subsection below covering the character classes applies to both BRE and ERE.

BRE and ERE work together. ERE adds ?, *, and |, and it removes the need to escape the metacharacters ( ) and { }, which are required in BRE. Furthermore, as long as the POSIX standard syntax for regular expressions is adhered to, there can be, and is, additional syntax to serve specific (yet POSIX compliant) applications. Although POSIX.2 leaves some implementation specifics undefined, BRE and ERE provide a "standard" which has since been adopted as the default syntax of many tools, where the choice of BRE or ERE modes is usually a supported option. For example, GNU grep has the following options: "grep -E" for ERE, and "grep -G" for BRE (the default), and "grep -p" for Perl regular expressions.

Perl regular expressions have become a de facto standard, having a rich and powerful set of atomic expressions. Perl has no "basic" "extended" level, where the ( ) and { } may or may not have literal meanings. They are always metacharacters, as they are in "extended" mode for POSIX. To get their "literal" meaning, you escape them. Other metacharacters are known to be literal or symbolic based on context alone. Perl offers much more functionality: "lazy" metacharacters, as they are in "extended" mode for POSIX. To get their meaning, you escape them. Other metacharacters have no "basic" "extended" level, where the ( ) and { } may or may not have literal meanings. They are always metacharacters, as they are in "extended" mode for POSIX. To get their "literal" meaning, you escape them. Other metacharacters are known to be literal or symbolic based on context alone. Perl offers much more functionality: "lazy" regular expressions, backtracking, named capture groups, and recursive patterns, all of which are powerful additions to POSIX BRE/ERE. (See lazy quantification below.)

POSIX basic and extended [edit]

In the POSIX standard, Basic Regular Syntax, BRE, requires that the metacharacters ( ) and { } be designated \( (\) and \( \), whereas Extended Regular Syntax, ERE, does not.

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor-, character-encoding-, and platform-specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, a.c matches &quot;abc&quot;, etc., but [a.c] matches only &quot;a&quot;, &quot;b&quot;, or &quot;c&quot;.</td>
</tr>
<tr>
<td>[ ]</td>
<td>A bracket expression. Matches a single character that is contained within the brackets. For example, [abc] matches &quot;a&quot;, &quot;b&quot;, or &quot;c&quot;. [a-z] specifies a range which matches any lowercase letter from &quot;a&quot; to &quot;z&quot;. These forms can be mixed: [abcx-z] matches &quot;a&quot;, &quot;b&quot;, &quot;c&quot;, &quot;x&quot;, &quot;y&quot;, or &quot;z&quot;, as does [a-cx-z]. The – character is treated as a literal character if it is the last or the first (after the ^, if present) character within the brackets: [abc–], [–abc]. Note that backslash escapes are not allowed. The ] character can be included in a bracket expression if it is the first (after the ^) character: []abc.</td>
</tr>
<tr>
<td>^</td>
<td>Matches a single character that is not contained within the brackets. For example, [^abc] matches any character other than &quot;a&quot;, &quot;b&quot;, or &quot;c&quot;. [^a-z] matches any single character that is not a lowercase letter from &quot;a&quot; to &quot;z&quot;. Likewise, literal characters and ranges can be mixed.</td>
</tr>
<tr>
<td>^</td>
<td>Matches the starting position within the string. In line-based tools, it matches the starting position of any line.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.</td>
</tr>
</tbody>
</table>
( )  Defines a marked subexpression. The string matched within the parentheses can be recalled later (see the next entry, \n). A marked subexpression is also called a block or capturing group. BRE mode requires \{ \}

\n  Matches what the rth marked subexpression matched, where n is a digit from 1 to 9. This construct is vaguely defined in the POSIX.2 standard. Some tools allow referencing more than nine capturing groups.

*  Matches the preceding element zero or more times. For example, ab+c matches "ac", "abc", "abbc", etc. [xy2]^+ matches "", "x", "y", "z", "zx", "zyx", "xyzy", and so on. (ab)* matches "", "ab", "abab", "ababab", and so on.

\(m, n\)  Matches the preceding element at least m and not more than n times. For example, a{3,5} matches only "aaa", "aaaa", and "aaaaa". This is not found in a few older instances of regular expressions. BRE mode requires {m, n}.

Examples:

- .at matches any three-character string ending with "at", including "hat", "cat", and "bat".
- [hc]at matches "hat" and "cat".
- [^b]at matches all strings matched by .at except "bat".
- ^[hc]at matches all strings matched by .at other than "hat" and "cat".
- [hc]^+at matches "hat" and "cat", but only at the beginning of the string or line.
- [hc]at[^] matches "hat" and "cat", but only at the end of the string or line.
- s.* matches any number of characters preceded by s, for example: "saw" and "seed".

POSIX extended  [edit]

The meaning of metacharacters escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, \( \) is now ( ) and \{ \} is now { }. Additionally, support is removed for \n backreferences and the following metacharacters are added:

<table>
<thead>
<tr>
<th>Metacharacter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Matches the preceding element zero or one time. For example, ab?c matches only &quot;ac&quot; or &quot;abc&quot;.</td>
</tr>
<tr>
<td>+</td>
<td>Matches the preceding element one or more times. For example, ab+c matches &quot;abc&quot;, &quot;abbc&quot;, &quot;abbbc&quot;, and so on, but not &quot;ac&quot;.</td>
</tr>
<tr>
<td>l</td>
<td>The choice (also known as alternation or set union) operator matches either the expression before or the expression after the operator. For example, abc</td>
</tr>
</tbody>
</table>

Examples:

- [hc]+at matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", and so on, but not "at".
- [hc]?at matches "hat", "cat", and "at".
- [hc]*at matches "hat", "cat", "hhat", "chat", "hcat", "cchchat", "at", and so on.
- cat|dog matches "cat" or "dog".

POSIX Extended Regular Expressions can often be used with modern Unix utilities by including the command line flag -E.

Character classes  [edit]

The character class is the most basic regular expression concept after a literal match. It makes one small sequence of characters match a larger set of characters. For example, [a-z] could stand for the alphabet, and \d could mean any digit. Character classes apply to both POSIX levels.

When specifying a range of characters, such as [a-z] (i.e. lowercase a to upper-case z), the computer's locale settings determine the contents by the numeric ordering of the character encoding. They could store digits in that sequence, or the ordering could be abc...zBC...Z, or aAbBcC...zZ. So the POSIX standard defines a character class, which will be known by the regular expression processor installed. Those definitions are in the following table:

<table>
<thead>
<tr>
<th>POSIX</th>
<th>Non-standard</th>
<th>Perl/Tcl</th>
<th>Vim</th>
<th>ASCII</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[:alnum:]</td>
<td>[A-Za-z0-9]</td>
<td>A-Za-z0-9</td>
<td>\w</td>
<td>Alphanumeric characters</td>
<td></td>
</tr>
<tr>
<td>[:word:]</td>
<td>\w</td>
<td>\w</td>
<td>[A-Za-z0-9 ]</td>
<td>Alphanumeric characters plus &quot;_&quot;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\W</td>
<td>\W</td>
<td>[*A-Za-z0-9 ]</td>
<td>Non-word characters</td>
<td></td>
</tr>
</tbody>
</table>
POSIX character classes can only be used within bracket expressions. For example, `[:upper:]ab` matches the uppercase letters and lowercase "a" and "b".  

An additional non-POSIX class understood by some tools is `[:word:]`, which is usually defined as `[:alnum:]` plus underscore. This reflects the fact that in many programming languages these are the characters that may be used in identifiers. The editor Vim further distinguishes `word` and `word-head` classes (using the notation `\w` and `\b`) since in many programming languages the characters that can begin an identifier are not the same as those that can occur in other positions.

Note that what the POSIX regular expression standards call character classes are commonly referred to as POSIX character classes in other regular expression flavors which support them. With most other regular expression flavors, the term character class is used to describe what POSIX calls bracket expressions.

### Standard Perl  

The Perl standard is still evolving in Perl 6, but the current set of symbols and syntax has become a de facto standard. 

Largely because of its expressive power, many other utilities and programming languages have adopted syntax similar to Perl's — for example, Java, JavaScript, Python, Ruby, Microsoft’s .NET Framework, and the W3C’s XML Schema all use regular expression syntax similar to Perl’s. Some languages and tools such as Boost and PHP support multiple regular expression flavors. Perl-derivative regular expression implementations are not identical and usually implement a subset of features found in Perl 5.0, released in 1994. Perl sometimes does incorporate features initially found in other languages, for example, Perl 5.10 implements syntactic extensions originally developed in PCRE and Python.[23]

### Lazy quantification  

Quantifiers match as many times as possible unless followed by `?`, when they match as few times as possible. If a quantifier is not followed by `?`, we say the quantifier is greedy. For example, consider the string

```
Another whale sighting occurred on <January 26>, <2004>.
```

To match (then display) only "<January 26>" and not ", <2004>" it is tempting to write `<.*>`. But there is more than one `,` and the expression can take the second one, and having both, still match, displaying "<January 26>, <2004>".

Because the `*` quantifier is greedy, it will consume as many characters as possible from the string, and "<January 26>, <2004>" has more characters than "<January 26>".

This problem can be avoided by specifying the text that is not to be matched: `<[^>]*>`, but modern regular expressions allow a quantifier to be specified as lazy. They put a question mark after the quantifier to make it lazy `<.*?>`. By using a lazy quantifier, the expression tries the minimal match first. Lazy matching may also be used to improve performance by avoiding backtracking in cases when the longest match is likely to be incorrect.[citation needed]
Many features found in modern regular expression libraries provide an expressive power that far exceeds the regular languages. For example, many implementations allow grouping subexpressions with parentheses and recalling the value they match in the same expression (backreferences). This means that, among other things, a pattern can match strings of repeated words like "papa" or "WikiWiki", called squares in formal language theory. The pattern for these strings is \( (.*) \1 \).

The language of squares is not regular, nor is it context-free, due to the pumping lemma. However, pattern matching with an unbounded number of back references, as supported by numerous modern tools, is still context sensitive. However, many tools, libraries, and engines that provide such constructions still use the term regular expression for their patterns. This has led to a nomenclature where the term regular expression has different meanings in formal language theory and pattern matching. For this reason, some people have taken to using the term regex or simply pattern to describe the latter. Larry Wall, author of the Perl programming language, writes in an essay about the design of Perl 6:

"Regular expressions [...] are only marginally related to real regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here. I will, however, generally call them "regexes" (or "regexen", when I'm in an Anglo-Saxon mood)."[15]

Fuzzy regular expressions [edit]

Variants of regular expressions can be used for working with text in natural language, when it is necessary to take into account possible typos and spelling variants. For example, the text "Julius Caesar" might be a fuzzy match for:

- Gaius Julius Caesar
- Yulius Cesar
- G. Jully Caezar

In such cases the mechanism implements some fuzzy string matching algorithm and possibly some algorithm for finding the similarity between text fragment and pattern.

This task is closely related to both full text search and named entity recognition.

Some software libraries work with fuzzy regular expressions:

- TRE – well-developed portable free project in C, which uses syntax similar to POSIX
- FREJ – open source project in Java with non-standard syntax (which utilizes prefix, Lisp-like notation), targeted to allow easy use of substitutions of inner matched fragments in outer blocks, but lacks many features of standard regular expressions.
- agrep – command-line utility (proprietary, but free for non-commercial usage).

Implementations and running times [edit]

There are at least three different algorithms that decide whether and how a given regular expression matches a string.

The oldest and fastest relies on a result in formal language theory that allows every nondeterministic finite automaton (NFA) to be transformed into a deterministic finite automaton (DFA). The DFA can be constructed explicitly and then run on the resulting input string one symbol at a time. Constructing the DFA for a regular expression of size \( m \) has the time and memory cost of \( O(2^m) \), but it can be run on a string of size \( n \) in time \( O(n) \). An alternative approach is to simulate the NFA directly, essentially building each DFA state on demand and then discarding it at the next step. This keeps the DFA implicit and avoids the exponential construction cost, but running cost rises to \( O(m \cdot n) \). The explicit approach is called the DFA algorithm and the implicit approach the NFA algorithm. Adding caching to the NFA algorithm is often called the "lazy DFA" algorithm, or just the DFA algorithm without making a distinction. These algorithms are fast, but using them for recalling grouped subexpressions, lazy quantification, and similar features is tricky.[24][25]

The third algorithm is to match the pattern against the input string by backtracking. This algorithm is commonly called NFA, but this terminology can be confusing. Its running time can be exponential, which simple implementations exhibit when matching against expressions like \( \texttt{(a|aa).+} \); that contain both alternation and unbounded quantification and force the algorithm to consider an exponentially increasing number of sub-cases. This behavior can cause a security problem called Regular expression Denial of Service.

Although backtracking implementations only give an exponential guarantee in the worst case, they provide much greater flexibility and expressive power. For example, any implementation which allows the use of backreferences, or

http://en.wikipedia.org/wiki/Regular_expression
Regular expression - Wikipedia, the free encyclopedia

 implements the various extensions introduced by Perl, must include some kind of backtracking. Some implementations try to provide the best of both algorithms by first running a fast DFA algorithm, and revert to a potentially slower backtracking algorithm only when a backreference is encountered during the match.

### Unicode [edit]

In theoretical terms, any token set can be matched by regular expressions as long as it is pre-defined. In terms of historical implementations, regular expressions were originally written to use ASCII characters as their token set though regular expression libraries have supported numerous other character sets. Many modern regular expression engines offer at least some support for Unicode. In most respects it makes no difference what the character set is, but some issues do arise when extending regular expressions to support Unicode.

- **Supported encoding.** Some regular expression libraries expect to work on some particular encoding instead of on abstract Unicode characters. Many of these require the UTF-8 encoding, while others might expect UTF-16, or UTF-32. In contrast, Perl and Java are agnostic on encodings, instead operating on decoded characters internally.

- **Supported Unicode range.** Many regular expression engines support only the Basic Multilingual Plane, that is, the characters which can be encoded with only 16 bits. Currently, only a few regular expression engines (e.g., Perl's and Java's) can handle the full 21-bit Unicode range.

- **Extending ASCII-oriented constructs to Unicode.** For example, in ASCII-based implementations, character ranges of the form [x-y] are valid wherever x and y have code points in the range [0x00,0x7F] and codepoint(x) ≤ codepoint(y). The natural extension of such character ranges to Unicode would simply change the requirement that the endpoints lie in [0x00,0x7F] to the requirement that they lie in [0,0x10FFFF]. However, in practice this is often not the case. Some implementations, such as that of gawk, do not allow character ranges to cross Unicode blocks. A range like [0x61,0x7F] is valid since both endpoints fall within the Basic Latin block, as is [0x0530,0x0560] since both endpoints fall within the Armenian block, but a range like [0x0061,0x0532] is invalid since it includes multiple Unicode blocks. Other engines, such as that of the Vim editor, allow block-crossing but the character values must not be more than 256 apart.[26]

- **Case insensitivity.** Some case-insensitivity flags affect only the ASCII characters. Other flags affect all characters. Some engines have two different flags, one for ASCII, the other for Unicode. Exactly which characters belong to the POSIX classes also varies.

- **Cousins of case insensitivity.** As ASCII has case distinction, case insensitivity became a logical feature in text searching. Unicode introduced alphabetic scripts without case like Devanagari. For these, case sensitivity is not applicable. For scripts like Chinese, another distinction seems logical: between traditional and simplified. In Arabic scripts, insensitivity to initial, medial, final, and isolated position may be desired. In Japanese, insensitivity between hiragana and katakana is sometimes useful.

- **Normalization.** Unicode has combining characters. Like old typewriters, plain letters can be followed by one of more non-spacing symbols (usually diacritics like accent marks) to form a single printing character, but also provides precomposed characters, i.e. characters that already include one or more combining characters. A sequence of a character + combining character should be matched with the identical single precomposed character. The process of standardizing sequences of characters + combining characters is called normalization. New control codes. Unicode introduced amongst others, byte order marks and text direction markers. These codes might have to be dealt with in a special way.

- **Introduction of character classes for Unicode blocks, scripts, and numerous other character properties.** Block properties are much less useful than script properties, because a block can have code points from several different scripts, and a script can have code points from several different blocks. In Perl and the java.util.regex library, properties of the form \p{X} or \p{Block=X} match characters in block X and \p{InX} or \p{Block=X} matches code points not in that block. Similarly, \p{Armenian}, \p{InArmenian}, or \p{Script=Armenian} matches any character in the Armenian script. In general, \p{X} matches any character with either the binary property X or the general category X. For example, \p{Lu} or \p{Uppercase_Letter} matches any uppercase letter. Binary properties that are not general categories include \p{White_Space}, \p{Alphabetic}, \p{Math}, and \p{Dash}. Examples of non-binary properties are \p{Bidi_Class=Right_to_Left}, \p{Word_Break=A_Letter}, and \p{Numeric_Value=10}.

### Uses [edit]

Regular expressions are useful in the production of syntax highlighting systems, data validation, and many other tasks.

While regular expressions would be useful on Internet search engines, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex. Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public. Notable exceptions: Google Code Search, Exalead.
Examples

This section may contain excessive, poor, or irrelevant examples. Please improve the article by adding more descriptive text and removing less pertinent examples. See Wikipedia's guide to writing better articles for further suggestions. (March 2012)

A regular expression is a string that is used to describe or match a set of strings according to certain syntax rules. The specific syntax rules vary depending on the specific implementation, programming language, or library in use. Additionally, the functionality of regex implementations can vary between versions.

Despite this variability, and because regular expressions can be difficult to both explain and understand without examples, interactive web sites for testing regular expressions are a useful resource for learning regular expressions by experimentation. This section provides a basic description of some of the properties of regular expressions by way of illustration.

The following conventions are used in the examples.

<table>
<thead>
<tr>
<th>Meta-character(s)</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Normally matches any character except a newline. Within square brackets the dot is literal.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/..../) { print &quot;$string1 has length &gt;= 5\n&quot;; }</td>
</tr>
<tr>
<td>( )</td>
<td>Groups a series of pattern elements to a single element. When you match a pattern within parentheses, you can use any of $1, $2, ... later to refer to the previously matched pattern.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/(\w+)/(\w+)/) { print &quot;We matched '$1' and '$2'\n&quot;; }</td>
</tr>
<tr>
<td>+</td>
<td>Matches the preceding pattern element one or more times.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\w+/) { print &quot;There are one or more consecutive letter &quot;&quot;s in $string1\n&quot;; }</td>
</tr>
<tr>
<td>?</td>
<td>Matches the preceding pattern element zero or one times.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\H.7e/) { print &quot;There is an 'H' and a 'e' separated by &quot;; print &quot;0-1 characters (Ex: He Hoe)\n&quot;; }</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>?</td>
<td>Modifies the *, +, 7 or {M,N}'d regex that comes before to match as few times as possible.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/(l.*?o)/) { print &quot;The non-greedy match with 'l' followed by zero or more characters is 'llo' rather than 'llo Wo'.\n&quot;; }</td>
</tr>
<tr>
<td>*</td>
<td>Matches the preceding pattern element zero or more times.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/el*o/) { print &quot;There is an 'e' followed by zero to many 'l's in $string1\n&quot;; }</td>
</tr>
<tr>
<td>{M,N}</td>
<td>Denotes the minimum M and the maximum N match count.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/l{1,2}/) { print &quot;There exists a substring with at least 1 and at most 2 l's in $string1\n&quot;; }</td>
</tr>
<tr>
<td>[\ldots]</td>
<td>Denotes a set of possible character matches.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/[aeiou]+/) { print &quot;$string1 contains one or more vowels.\n&quot;; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/Hello</td>
</tr>
<tr>
<td>\b</td>
<td>Matches a zero-width boundary between a word-class character (see next) and either a non-word class character or an edge.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/llo\b/) { print &quot;There is a word that ends with 'llo'\n&quot;; }</td>
</tr>
<tr>
<td>\w</td>
<td>Matches an alphanumeric character, including &quot;<em>&quot;; same as [A-Za-z0-9</em>] in ASCII. In Unicode, same as <code>\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}</code>. where the Alphabetic property contains more than just Letters, and the Decimal_Number property contains more than ([0-9]).</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\d/) { print &quot;There is at least one alphanumeric &quot;; print &quot;character in $string1\n&quot;; }</td>
</tr>
<tr>
<td>\W</td>
<td>Matches a non-alphanumeric character, excluding &quot;<em>&quot;; same as [^A-Za-z0-9</em>] in ASCII, and <code>\p{Alphabetic}\p{GC=Mark}</code>.</td>
<td>$string1 = &quot;Hello World\n&quot;; if ($string1 =~ m/\W/) { print &quot;The space between Hello and\n&quot;; }</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
<td>Code Example</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| `\p{GC=Decimal_Number}` | Matches a digit; same as `[0-9]` in ASCII; in Unicode, same as the `\p{Digit}` or `\p{GC=Decimal_Number}` property, which itself the same as the `\p{Numeric_Type=Decimal}` property. | `$string1 = "99 bottles of beer on the wall."; if ($string1 =~ m/\d+/) { print "$1 is the first number in '$string1'
"; }` |
| `\s`         | Matches a whitespace character, which in ASCII are tab, line feed, form feed, carriage return, and space; in Unicode, also matches no-break spaces, next line, and the variable-width spaces (amongst others). | `$string1 = "Hello World\n"; if ($string1 =~ m/s.*\s/ { print "There are TWO whitespace characters, which may"; print " be separated by other characters, in $string1"; }` |
| `\S`         | Matches anything BUT a whitespace.                                            | `$string1 = "Hello World\n"; if ($string1 =~ m/\S.*\S/ { print "There are TWO non-whitespace characters, which may be separated by other characters, in $string1"; }` |
| `\d`         | Matches a digit; same as `[0-9]` in ASCII; in Unicode, same as the `\p{Digit}` or `\p{GC=Decimal_Number}` property, which itself the same as the `\p{Numeric_Type=Decimal}` property. | `$string1 = "99 bottles of beer on the wall."; if ($string1 =~ m/\d+/) { print "$1 is the first number in '$string1'
"; }` |
| `\D`         | Matches a non-digit; same as `[^0-9]` in ASCII or `\P{Digit}` in Unicode.    | `$string1 = "Hello World\n"; if ($string1 =~ m/\D/) { print "There is at least one character in $string1"; print " that is not a digit."\n"; }` |
| `^`          | Matches the beginning of a line or string.                                   | `$string1 = "Hello World\n"; if ($string1 =~ m/^He/) { print "$string1 starts with the characters 'He'\n"; }` |
| `$`          | Matches the end of a line or string.                                         | `$string1 = "Hello World\n"; if ($string1 =~ m/rld\$) { print "$string1 is a line or string that ends with 'rld'\n"; }` |
| `\A`         | Matches the beginning of a string (but not an internal line).               | `$string1 = "Hello World\n"; if ($string1 =~ m/\AH/) { print "$string1 is a string that starts with 'H'\n"; }` |
| `\z`         | Matches the end of a string (but not an internal line).[^1]                   | `$string1 = "Hello World\n"; if ($string1 =~ m/d\n\z/) { print "$string1 is a string that ends with 'n\n'"; }` |
[^...]

Matches every character except the ones inside brackets.

See also [edit]

- Comparison of regular expression engines
- Extended Backus–Naur Form
- List of regular expression software — applications which support regular expressions
- Regular tree grammar
- Thompson's construction algorithm — converts a regular expression into an equivalent NFA

Notes [edit]

2. ^ a b Johnson et al. 1968.
3. ^ a b Thompson 1968.
15. ^ a b Wall (2002).
27. ^ a b "UTS#18 on Unicode Regular Expressions, Annex A: Character Blocks". Retrieved 2010-02-05.
28. ^ The character 'm' is not always required to specify a Perl match operation. For example, m/^[a-abc]$/ could also be rendered as /^(a|bc|). The 'm' is only necessary if the user wishes to specify a match operation without using a forward-slash as the regex delimiter. Sometimes it is useful to specify an alternate regex delimiter in order to avoid "delimiter collision". See 'perl documentation' for more details.
29. ^ e.g., see Java in a Nutshell — Page 213, Python Scripting for Computational Science — Page 320, Programming PHP — Page 106.
30. ^ Note that all the if statements return a TRUE value.

References [edit]
- Laurikari, Ville (2009). "TRE library 0.7.6".

**External links** [edit]

**Standards** [edit]

**Language references** [edit]
- Java Tutorials: Regular Expressions
- Perl Regular Expressions documentation
- VBScript and Regular Expressions
- .NET Framework Regular Expressions
- JavaScript Regular Expressions Chapter and RegExp Object Reference at the Mozilla Developer Center

**Links** [edit]
- Regular Expressions at DMOZ
- Pattern matching tools and libraries
- Search for interactive regular expression testers (testers generally require browser extensions beyond the minimum requirements for viewing Wikipedia)

**Other** [edit]
- Structural Regular Expressions, by Rob Pike
Implementing Regular Expressions, Russ Cox, including references to assist in reading Thompson (1968):

- IBM 7094 Cheat Sheet, Russ Cox, November 2000 (updated January 2007)
- Implementing Regular Expressions, CS 121 Extra Lecture, November 17, 2000

Categories: Automata theory | Formal languages | Pattern matching | Programming constructs
| Regular expressions