regexp

# Package regexp go1.15.2 Latest

Published: **Sep 9, 2020** | License: BSD-3-Clause | Standard library



## Overview

Package regexp implements regular expression search.

The syntax of the regular expressions accepted is the same general syntax used by Perl, Python, and other languages. More precisely, it is the syntax accepted by RE2 and described at https://golang.org/s/re2syntax, except for \C. For an overview of the syntax, run

```
go doc regexp/syntax
```

The regexp implementation provided by this package is guaranteed to run in time linear in the size of the input. (This is a property not guaranteed by most open source implementations of regular expressions.) For more information about this property, see

```
https://swtch.com/~rsc/regexp/regexp1.html
```

or any book about automata theory.

All characters are UTF-8-encoded code points.

There are 16 methods of Regexp that match a regular expression and identify the matched text. Their names are matched by this regular expression:

```
Find(All)?(String)?(Submatch)?(Index)?
```

If 'All' is present, the routine matches successive non-overlapping matches of the entire expression. Empty matches abutting a preceding match are ignored. The return value is a slice containing the successive return values of the corresponding non-'All' routine. These routines take an extra integer argument, n. If n >= 0, the function returns at most n matches/submatches; otherwise, it returns all of them.

If 'String' is present, the argument is a string; otherwise it is a slice of bytes; return values are adjusted as appropriate.

If 'Submatch' is present, the return value is a slice identifying the successive submatches of the expression. Submatches are matches of parenthesized subexpressions (also known as capturing groups) within the regular expression, numbered from left to right in order of opening parenthesis. Submatch 0 is the match of the entire expression, submatch 1 the match of the first parenthesized subexpression, and so on.

If 'Index' is present, matches and submatches are identified by byte index pairs within the input string: result[2*n:2*n+1] identifies the indexes of the nth submatch. The pair for n==0 identifies the match of the entire expression. If 'Index' is not present, the match is identified by the text of the match/submatch. If an index is negative or text is nil, it means that subexpression did not match any string in the input. For 'String' versions an empty string means either no match or an empty match.

There is also a subset of the methods that can be applied to text read from a RuneReader:

```
MatchReader, FindReaderIndex, FindReaderSubmatchIndex
```

This set may grow. Note that regular expression matches may need to examine text beyond the text returned by a match, so the methods that match text from a RuneReader may read arbitrarily far into the input before returning.

(There are a few other methods that do not match this pattern.)

## func Match

```
func Match(pattern string, b []byte) (matched bool, err error)
```

Match reports whether the byte slice b contains any match of the regular expression pattern. More complicated queries need to use Compile and the full Regexp interface.

## func MatchReader

```
func MatchReader(pattern string, r io.RuneReader) (matched bool, err error)
```

MatchReader reports whether the text returned by the RuneReader contains any match of the regular expression pattern. More complicated queries need to use Compile and the full Regexp interface.

## func MatchString

```
func MatchString(pattern string, s string) (matched bool, err error)
```

MatchString reports whether the string s contains any match of the regular expression pattern. More complicated queries need to use Compile and the full Regexp interface.

## func QuoteMeta

```
func QuoteMeta(s string) string
```

QuoteMeta returns a string that escapes all regular expression metacharacters inside the argument text; the returned string is a regular expression matching the literal text.

## type Regexp

```
type Regexp struct {
    // contains filtered or unexported fields
}
```

Regexp is the representation of a compiled regular expression. A Regexp is safe for concurrent use by multiple goroutines, except for configuration methods, such as Longest.

## func Compile

```
func Compile(expr string) (*Regexp, error)
```

Compile parses a regular expression and returns, if successful, a Regexp object that can be used to match against text.

When matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses the one that a backtracking search would have found first. This so-called leftmost-first matching is the same semantics that Perl, Python, and other implementations use, although this package implements it without the expense of backtracking. For POSIX leftmost-longest matching, see CompilePOSIX.

## func CompilePOSIX

```
func CompilePOSIX(expr string) (*Regexp, error)
```

CompilePOSIX is like Compile but restricts the regular expression to POSIX ERE (egrep) syntax and changes the match semantics to leftmost-longest.

That is, when matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses a match that is as long as possible. This so-called leftmost-longest matching is the same semantics that early regular expression implementations used and that POSIX specifies.

However, there can be multiple leftmost-longest matches, with different submatch choices, and here this package diverges from POSIX. Among the possible leftmost-longest matches, this package chooses the one that a backtracking search would have found first, while POSIX specifies that the match be chosen to maximize the length of the first subexpression, then the second, and so on from left to right. The POSIX rule is computationally prohibitive and not even well-defined. See https://swtch.com/~rsc/regexp/regexp2.html#posix for details.

## func MustCompile

```
func MustCompile(str string) *Regexp
```

MustCompile is like Compile but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

## func MustCompilePOSIX

```
func MustCompilePOSIX(str string) *Regexp
```

MustCompilePOSIX is like CompilePOSIX but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

## func (*Regexp) Copy

```
func (re *Regexp) Copy() *Regexp
```

Copy returns a new Regexp object copied from re. Calling Longest on one copy does not affect another.

Deprecated: In earlier releases, when using a Regexp in multiple goroutines, giving each goroutine its own copy helped to avoid lock contention. As of Go 1.12, using Copy is no longer necessary to avoid lock contention. Copy may still be appropriate if the reason for its use is to make two copies with different Longest settings.

## func (*Regexp) Expand

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, match []int) []byte
```

Expand appends template to dst and returns the result; during the append, Expand replaces variables in the template with corresponding matches drawn from src. The match slice should have been returned by FindSubmatchIndex.

In the template, a variable is denoted by a substring of the form $name or ${name}, where name is a non-empty sequence of letters, digits, and underscores. A purely numeric name like $1 refers to the submatch with the corresponding index; other names refer to capturing parentheses named with the (?P<name>...) syntax. A reference to an out of range or unmatched index or a name that is not present in the regular expression is replaced with an empty slice.

In the $name form, name is taken to be as long as possible: $1x is equivalent to ${1x}, not ${1}x, and, $10 is equivalent to ${10}, not ${1}0.

To insert a literal $ in the output, use $$ in the template.

## func (*Regexp) ExpandString

```
func (re *Regexp) ExpandString(dst []byte, template string, src string, match []int)
```

ExpandString is like Expand but the template and source are strings. It appends to and returns a byte slice in order to give the calling code control over allocation.

## func (*Regexp) Find

```
func (re *Regexp) Find(b []byte) []byte
```

Find returns a slice holding the text of the leftmost match in b of the regular expression. A return value of nil indicates no match.

## func (*Regexp) FindAll

```
func (re *Regexp) FindAll(b []byte, n int) [][]byte
```

FindAll is the 'All' version of Find; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllIndex

```
func (re *Regexp) FindAllIndex(b []byte, n int) [][]int
```

FindAllIndex is the 'All' version of FindIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllString

```
func (re *Regexp) FindAllString(s string, n int) []string
```

FindAllString is the 'All' version of FindString; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllStringIndex

```
func (re *Regexp) FindAllStringIndex(s string, n int) [][]int
```

FindAllStringIndex is the 'All' version of FindStringIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllStringSubmatch

```
func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]string
```

FindAllStringSubmatch is the 'All' version of FindStringSubmatch; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllStringSubmatchIndex

```
func (re *Regexp) FindAllStringSubmatchIndex(s string, n int) [][]int
```

FindAllStringSubmatchIndex is the 'All' version of FindStringSubmatchIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllSubmatch

```
func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte
```

FindAllSubmatch is the 'All' version of FindSubmatch; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindAllSubmatchIndex

```
func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]int
```

FindAllSubmatchIndex is the 'All' version of FindSubmatchIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindIndex

```
func (re *Regexp) FindIndex(b []byte) (loc []int)
```

FindIndex returns a two-element slice of integers defining the location of the leftmost match in b of the regular expression. The match itself is at b[loc[0]:loc[1]]. A return value of nil indicates no match.

## func (*Regexp) FindReaderIndex

```
func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int)
```

FindReaderIndex returns a two-element slice of integers defining the location of the leftmost match of the regular expression in text read from the RuneReader. The match text was found in the input stream at byte offset loc[0] through loc[1]-1. A return value of nil indicates no match.

## func (*Regexp) FindReaderSubmatchIndex

```
func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) []int
```

FindReaderSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression of text read by the RuneReader, and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindString

```
func (re *Regexp) FindString(s string) string
```

FindString returns a string holding the text of the leftmost match in s of the regular expression. If there is no match, the return value is an empty string, but it will also be empty if the regular expression successfully matches an empty string. Use FindStringIndex or FindStringSubmatch if it is necessary to distinguish these cases.

## func (*Regexp) FindStringIndex

```
func (re *Regexp) FindStringIndex(s string) (loc []int)
```

FindStringIndex returns a two-element slice of integers defining the location of the leftmost match in s of the regular expression. The match itself is at s[loc[0]:loc[1]]. A return value of nil indicates no match.

## func (*Regexp) FindStringSubmatch

```
func (re *Regexp) FindStringSubmatch(s string) []string
```

FindStringSubmatch returns a slice of strings holding the text of the leftmost match of the regular expression in s and the matches, if any, of its subexpressions, as defined by the 'Submatch' description in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindStringSubmatchIndex

```
func (re *Regexp) FindStringSubmatchIndex(s string) []int
```

FindStringSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression in s and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindSubmatch

```
func (re *Regexp) FindSubmatch(b []byte) [][]byte
```

FindSubmatch returns a slice of slices holding the text of the leftmost match of the regular expression in b and the matches, if any, of its subexpressions, as defined by the 'Submatch' descriptions in the package comment. A return value of nil indicates no match.

## func (*Regexp) FindSubmatchIndex

```
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

FindSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression in b and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

## func (*Regexp) LiteralPrefix

```
func (re *Regexp) LiteralPrefix() (prefix string, complete bool)
```

LiteralPrefix returns a literal string that must begin any match of the regular expression re. It returns the boolean true if the literal string comprises the entire regular expression.

## func (*Regexp) Longest

```
func (re *Regexp) Longest()
```

Longest makes future searches prefer the leftmost-longest match. That is, when matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses a match that is as long as possible. This method modifies the Regexp and may not be called concurrently with any other methods.

## func (*Regexp) Match

```
func (re *Regexp) Match(b []byte) bool
```

Match reports whether the byte slice b contains any match of the regular expression re.

## func (*Regexp) MatchReader

```
func (re *Regexp) MatchReader(r io.RuneReader) bool
```

MatchReader reports whether the text returned by the RuneReader contains any match of the regular expression re.

## func (*Regexp) MatchString

```
func (re *Regexp) MatchString(s string) bool
```

MatchString reports whether the string s contains any match of the regular expression re.

## func (*Regexp) NumSubexp

```
func (re *Regexp) NumSubexp() int
```

NumSubexp returns the number of parenthesized subexpressions in this Regexp.

## func (*Regexp) ReplaceAll

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
```

ReplaceAll returns a copy of src, replacing matches of the Regexp with the replacement text repl. Inside repl, $ signs are interpreted as in Expand, so for instance $1 represents the text of the first submatch.

## func (*Regexp) ReplaceAllFunc

```
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []byte) []byte
```

ReplaceAllFunc returns a copy of src in which all matches of the Regexp have been replaced by the return value of function repl applied to the matched byte slice. The replacement returned by repl is substituted directly, without using Expand.

## func (*Regexp) ReplaceAllLiteral

```
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
```

ReplaceAllLiteral returns a copy of src, replacing matches of the Regexp with the replacement bytes repl. The replacement repl is substituted directly, without using Expand.

## func (*Regexp) ReplaceAllLiteralString

```
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
```

ReplaceAllLiteralString returns a copy of src, replacing matches of the Regexp with the replacement string repl. The replacement repl is substituted directly, without using Expand.

## func (*Regexp) ReplaceAllString

```
func (re *Regexp) ReplaceAllString(src, repl string) string
```

ReplaceAllString returns a copy of src, replacing matches of the Regexp with the replacement string repl. Inside repl, $ signs are interpreted as in Expand, so for instance $1 represents the text of the first submatch.

## func (*Regexp) ReplaceAllStringFunc

```
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string) string) string
```

ReplaceAllStringFunc returns a copy of src in which all matches of the Regexp have been replaced by the return value of function repl applied to the matched substring. The replacement returned by repl is substituted directly, without using Expand.

## func (*Regexp) Split

```
func (re *Regexp) Split(s string, n int) []string
```

Split slices s into substrings separated by the expression and returns a slice of the substrings between those expression matches.

The slice returned by this method consists of all the substrings of s not contained in the slice returned by FindAllString. When called on an expression that contains no metacharacters, it is equivalent to strings.SplitN.

Example:

```
s := regexp.MustCompile("a*").Split("abaabaccadaaae", 5)
// s: ["", "b", "b", "c", "cadaaae"]
```

The count determines the number of substrings to return:

```
n > 0: at most n substrings; the last substring will be the unsplit remainder.
n == 0: the result is nil (zero substrings)
n < 0: all substrings
```

## func (*Regexp) String

```
func (re *Regexp) String() string
```

String returns the source text used to compile the regular expression.

## func (*Regexp) SubexpIndex

```
func (re *Regexp) SubexpIndex(name string) int
```

SubexpIndex returns the index of the first subexpression with the given name, or -1 if there is no subexpression with that name.

Note that multiple subexpressions can be written using the same name, as in (?P<bob>a+)(?P<bob>b+), which declares two subexpressions named "bob". In this case, SubexpIndex returns the index of the leftmost such subexpression in the regular expression.

## func (*Regexp) SubexpNames

```
func (re *Regexp) SubexpNames() []string
```

SubexpNames returns the names of the parenthesized subexpressions in this Regexp. The name for the first sub-expression is names[1], so that if m is a match slice, the name for m[i] is SubexpNames()

[i]. Since the Regexp as a whole cannot be named, names[0] is always the empty string. The slice should not be modified.