Package 're2'

May 9, 2026

Type Package

Title R Interface to Google RE2 (C++) Regular Expression Library

Version 0.1.4

Date 2025-01-11

Description Pattern matching, extraction, replacement and other string processing operations using Google's RE2 <https://github.com/google/re2> regular-expression engine. Consistent interface (similar to 'stringr'). RE2 uses finite-automata based techniques, and offers a fast and safe alternative to backtracking regular-expression engines like those used in 'stringr', 'stringi' and other PCRE implementations.

License MIT + file LICENSE

Imports Rcpp (>= 1.0.8.3)

LinkingTo Rcpp

URL https://github.com/girishji/re2

BugReports https://github.com/girishji/re2/issues

Encoding UTF-8

RoxygenNote 7.3.2

Suggests knitr, rmarkdown

VignetteBuilder knitr

NeedsCompilation yes

Author Girish Palya [aut, cre],
RE2 developers [ctb] (RE2 library),
Google Inc. [ctb, cph] (RE2 library)

Maintainer Girish Palya <girishji@gmail.com>

Repository CRAN

Date/Publication 2025-01-19 22:40:02 UTC

Contents

`re2_count` *Count the number of matches in a string*

Description

Vectorized over string and pattern. Match against a string using a regular expression and return the count of matches.

Usage

```
re2_count(string, pattern)
```

Arguments

`string`	A character vector, or an object which can be coerced to one.
`pattern`	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). See `re2_regexp` for available options. See re2_syntax for regular expression syntax.

Value

An integer vector.

See Also

`re2_regexp` for options to regular expression, re2_syntax for regular expression syntax.

Examples

```
color <- c("yellowgreen", "steelblue", "goldenrod", "forestgreen")
re2_count(color, "e")
re2_count(color, "r")

# Regular expression vs literal string
re2_count(c("..", "a...", "foo.b"), ".")
re2_count(c("..", "a...", "foo.b"), re2_regexp(".", literal = TRUE))
```

`re2_detect` *Find the presence of a pattern in string(s)*

Description

Equivalent to grepl(pattern, x). Vectorized over string and pattern. For the equivalent of grep(pattern, x) see `re2_which`.

Usage

```
re2_detect(string, pattern)
```

Arguments

`string`	A character vector, or an object which can be coerced to one.
`pattern`	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). See `re2_regexp` for available options. See re2_syntax for regular expression syntax.

Value

A logical vector. TRUE if match is found, FALSE if not.

See Also

`re2_regexp` for options to regular expression, re2_syntax for regular expression syntax, and `re2_match` to extract matched groups.

Examples

```
## Character vector input
s <- c("barbazbla", "foobar", "not present here ")
pat <- "(foo)|(bar)baz"
re2_detect(s, pat)

## Use precompiled regexp
re <- re2_regexp("(foo)|(bAR)baz", case_sensitive = FALSE)
re2_detect(s, re)
```

re2_extract_replace *Extract with substitutions*

Description

Like re2_replace, except that if the pattern matches, "rewrite" string is returned with substitutions. The non-matching portions of "text" are ignored.

Difference between re2_extract_replace and re2_replace:

```
> re2_extract_replace("bunny@wunnies.pl", "(.*)@([^.]*)", "\2!\1")
[1] "wunnies!bunny"

> re2_replace("bunny@wunnies.pl", "(.*)@([^.]*)", "\2!\1")
[1] "wunnies!bunny.pl"
```

"\1" and "\2" are names of capturing subgroups.

Vectorized over string and pattern.

Usage

```
re2_extract_replace(string, pattern, rewrite)
```

Arguments

string	A character vector, or an object which can be coerced to one.
pattern	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). For re2_replace_all this can also be a named vector (c(pattern1 = rewrite1)), in order to perform multiple replacements in each element of string. See re2_regexp for available options. See re2_syntax for regular expression syntax.
rewrite	Rewrite string. Backslash-escaped digits (\1 to \9) can be used to insert text matching corresponding parenthesized group from the pattern. \0 refers to the entire matching text.

Value

A character vector with extractions.

See Also

re2_regexp for options to regular expression, re2_syntax for regular expression syntax. See re2_replace and re2_replace_all to replace pattern in place.

Examples

```
# Returns extracted string with substitutions
re2_extract_replace(
  "bunny@wunnies.pl",
  "(.*)@([^.]*)",
  "\\2!\\1"
)

# Case insensitive
re2_extract_replace(
  "BUNNY@wunnies.pl",
  re2_regexp("(b.*)@([^.]*)", case_sensitive = FALSE),
  "\\2!\\1"
)

# Max submatch too large (1 match group, 2 submatches needed).
#   Replacement fails and empty string is returned.
re2_extract_replace("foo", "f(o+)", "\\1\\2")
```

`re2_get_options` *Retrieve options*

Description

`re2_get_options` returns a list of all options from a RE2 object (internal representation of compiled regexp).

Usage

```
re2_get_options(re2ptr)
```

Arguments

`re2ptr`	The value obtained from call to `re2_regexp`.

Value

A list of options and their values.

See Also

`re2_regexp`.

re2_locate *Locate the start and end of pattern in a string*

Description

Vectorized over string and pattern. For matches of 0 length (ex. spatial patterns like "$") end will be one character greater than beginning.

Usage

```
re2_locate(string, pattern)

re2_locate_all(string, pattern)
```

Arguments

string	A character vector, or an object which can be coerced to one.
pattern	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). See re2_regexp for available options. See re2_syntax for regular expression syntax.

Value

re2_locate returns an integer matrix, and re2_locate_all returns a list of integer matrices.

See Also

re2_regexp for options to regular expression, re2_syntax for regular expression syntax.

Examples

```
color <- c("yellowgreen", "steelblue", "goldenrod", "forestgreen")

re2_locate(color, "$")
re2_locate(color, "l")
re2_locate(color, "e")

# String length can be a multiple of pattern length
re2_locate(color, c("l(l|d)?", "st"))

# Locate all occurrences
re2_locate_all(color, "l")
re2_locate_all(color, "e")

# Locate all characters
re2_locate_all(color, ".")
```

re2_match *Extract matched groups from a string*

Description

Vectorized over string and pattern. Match against a string using a regular expression and extract matched substrings. re2_match extracts first matched substring, and re2_match_all extracts all matches.

Matching regexp "(foo)|(bar)baz" on "barbazbla" will return submatches '.0' = "barbaz", '.1' = NA, and '.2' = "bar". '.0' is the entire matching text. '.1' is the first group, and so on. Groups can also be named.

Usage

```
re2_match(string, pattern, simplify = TRUE)

re2_match_all(string, pattern)
```

Arguments

string	A character vector, or an object which can be coerced to one.
pattern	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). See re2_regexp for available options. See re2_syntax for regular expression syntax.
simplify	If TRUE, the default, returns a character matrix. If FALSE, returns a list. Not applicable to re2_match_all.

Value

In case of re2_match a character matrix. First column is the entire matching text, followed by one column for each capture group. If simplify is FALSE, returns a list of named character vectors.
In case of re2_match_all, returns a list of character matrices.

See Also

re2_regexp for options to regular expression, re2_syntax for regular expression syntax.

Examples

```
## Substring extraction
strings <- c("barbazbla", "foobar")
pattern <- "(foo)|(?P<TestGroup>bar)baz"

re2_match(strings, pattern)
result <- re2_match(strings, pattern)
```

```
is.matrix(result)

re2_match(strings, pattern, simplify = FALSE)
result <- re2_match(strings, pattern, simplify = FALSE)
is.list(result)

## Compile regexp
re <- re2_regexp("(foo)|(BaR)baz", case_sensitive = FALSE)
re2_match(strings, re)

strings <- c(
  "Home: 743 733 5365", "373-733-5753 ", "foobar",
  "733.335.3457 and Work: 573-433-7577 "
)
re <- re2_regexp("([0-9]{3})[- .]([0-9]{3})[- .]([0-9]{4})")
re2_match(strings, re)

## Vectorized over patterns
re2_match(strings, c(re, "53 $", "^foo", re))

## Match all occurances, not just the first
re2_match_all(strings, re)
re2_match_all("ruby:1234 68 red:92 blue:", "(\\w+):(\\d+)")

## Vectorized over patterns (matching all occurances)
re2_match_all(strings, c(re, "53 $", "^foo", re))
```

`re2_regexp` *Compile regular expression pattern*

Description

`re2_regexp` compiles a character string containing a regular expression and returns a pointer to the object.

Usage

```
re2_regexp(pattern, ...)
```

Arguments

`pattern`	Character string containing a regular expression.
`...`	Options, which are (defaults in parentheses):
`encoding`	("`UTF8`") String and pattern are UTF-8; Otherwise "`Latin1`".
`posix_syntax`	(`FALSE`) Restrict regexps to POSIX egrep syntax.
`longest_match`	(`FALSE`) Search for longest match, not first match.
`max_mem`	(see below) Approx. max memory footprint of RE2 C++ object.
`literal`	(`FALSE`) Interpret pattern as literal, not regexp.

never_nl	(FALSE) Never match \n, even if it is in regexp.
dot_nl	(FALSE) Dot matches everything including new line.
never_capture	(FALSE) Parse all parens as non-capturing.
case_sensitive	(TRUE) Match is case-sensitive (regexp can override with (?i) unless in posix_syntax mode).

The following options are only consulted when posix_syntax=TRUE. When posix_syntax=FALSE, these features are always enabled and cannot be turned off; to perform multi-line matching in that case, begin the regexp with (?m).

perl_classes	(FALSE) Allow Perl's \d \s \w \D \S \W.
word_boundary	(FALSE) Allow Perl's \b \B (word boundary and not).
one_line	(FALSE) ^ and $ only match beginning and end of text.

The max_mem option controls how much memory can be used to hold the compiled form of the regexp and its cached DFA graphs (DFA: The execution engine that implements Deterministic Finite Automaton search). Default is 8MB.

Value

Compiled regular expression.

Regexp Syntax

RE2 regular expression syntax is similar to Perl's with some of the more complicated things thrown away. In particular, backreferences and generalized assertions are not available, nor is \Z.

See re2_syntax for the syntax supported by RE2, and a comparison with PCRE and PERL regexps.

For those not familiar with Perl's regular expressions, here are some examples of the most commonly used extensions:

"hello (\w+) world"	–	\w matches a "word" character.
"version (\d+)"	–	\d matches a digit.
"hello\s+world"	–	\s matches any whitespace character.
"\b(\w+)\b"	–	\b matches non-empty string at word boundary.
"(?i)hello"	–	(?i) turns on case-insensitive matching.
"/*(.*?)*/"	–	.*? matches . minimum no. of times possible.

The double backslashes are needed when writing R string literals. However, they should NOT be used when writing raw string literals:

r"(hello (\w+) world)"	–	\w matches a "word" character.
r"(version (\d+))"	–	\d matches a digit.
r"(hello\s+world)"	–	\s matches any whitespace character.
r"(\b(\w+)\b)"	–	\b matches non-empty string at word boundary.
r"((?i)hello)"	–	(?i) turns on case-insensitive matching.
r"(/*(.*?)*/)"	–	.*? matches . minimum no. of times possible.

When using UTF-8 encoding, case-insensitive matching will perform simple case folding, not full case folding.

See Also

re2_syntax has regular expression syntax.

Examples

```
re2p <- re2_regexp("hello world")
stopifnot(mode(re2p) == "externalptr")

## UTF-8 and matching interface
# By default, pattern and input text are interpreted as UTF-8.
# The Latin1 option causes them to be interpreted as Latin-1.
x <- "fa\xE7ile"
Encoding(x) <- "latin1"
re2_detect(x, re2_regexp("fa\xE7", encoding = "Latin1"))

## Case insensitive
re2_detect("fOobar ", re2_regexp("Foo", case_sensitive = FALSE))

## Literal string (as opposed to regular expression)
## Matches only when 'literal' option is TRUE
re2_detect("foo\\$bar", re2_regexp("foo\\$b", literal = TRUE))
re2_detect("foo\\$bar", re2_regexp("foo\\$b", literal = FALSE))

## Use of never_nl
re <- re2_regexp("(abc(.|\n)*def)", never_nl = FALSE)
re2_match("abc\ndef\n", re)
re <- re2_regexp("(abc(.|\n)*def)", never_nl = TRUE)
re2_match("abc\ndef\n", re)
```

---

re2_replace *Replace matched pattern in string*

Description

re2_replace replaces the first match of "pattern" in "string" with "rewrite" string.

```
re2_replace("yabba dabba doo", "b+", "d")
```

will result in "yada dabba doo".

re2_replace_all replaces successive non-overlapping occurrences of "pattern" in "text" with "rewrite" string, or performs multiple replacements on each element of string.

```
re2_replace_all("yabba dabba doo", "b+", "d")
re2_replace_all(c("one", "two"), c("one" = "1", "1" = "2", "two" = "2"))
```

will result in "yada dada doo".
Replacements are not subject to re-matching. Because re2_replace_all only replaces non-overlapping matches, replacing "ana" within "banana" makes only one replacement, not two.

Vectorized over string and pattern.

Usage

```
re2_replace(string, pattern, rewrite)

re2_replace_all(string, pattern, rewrite = "")
```

Arguments

string	A character vector, or an object which can be coerced to one.
pattern	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). For `re2_replace_all` this can also be a named vector (`c(pattern1 = rewrite1)`), in order to perform multiple replacements in each element of string. See `re2_regexp` for available options. See re2_syntax for regular expression syntax.
rewrite	Rewrite string. Backslash-escaped digits (\1 to \9) can be used to insert text matching corresponding parenthesized group from the pattern. \0 refers to the entire matching text.

Value

A character vector with replacements.

See Also

`re2_regexp` for options to regular expression, re2_syntax for regular expression syntax.

Examples

```
string <- c("yabba dabba doo", "famabbb sb")
re2_replace(string, "b+", "d")
re2_replace_all(string, "b+", "d")

# Rearrange matching groups in replaced string
re2_replace(
  "boris@kremvax.ru",
  "(.*)@([^.]*)", "\\2!\\1"
)

# Use complied pattern
string <- "the quick brown fox jumps over the lazy dogs."
re <- re2_regexp("(qu|[b-df-hj-np-tv-z]*)([a-z]+)")
rewrite <- "\\2\\1ay"
re2_replace(string, re, rewrite)
re2_replace_all(string, re, rewrite)

string <- "abcd.efghi@google.com"
re <- re2_regexp("\\w+")
rewrite <- "\\0-NOSPAM"
re2_replace(string, re, rewrite)
```

```
re2_replace_all(string, re, rewrite)

string <- "aba\naba"
re <- re2_regexp("a.*a")
rewrite <- "(\\0)"
re2_replace(string, re, rewrite)
re2_replace_all(string, re, rewrite)

# Vectorize string and pattern
string <- c("abababababab", "bbbbbb", "bbbbbb", "aaaaa")
pattern <- c("b", "b+", "b*", "b*")
rewrite <- "bb"
re2_replace(string, pattern, rewrite)
re2_replace_all(string, pattern, rewrite)
```

`re2_split` *Split string based on pattern*

Description

Vectorized over string and pattern.

Usage

```
re2_split(string, pattern, simplify = FALSE, n = Inf)
```

Arguments

`string`	A character vector, or an object which can be coerced to one.
`pattern`	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). See `re2_regexp` for available options. See re2_syntax for regular expression syntax.
`simplify`	If FALSE, the default, return a list of string vectors. If TRUE, return a string matrix.
`n`	Number of string pieces to return. Default (Inf) returns all.

Value

A list of string vectors or a string matrix. See option.

See Also

`re2_regexp` for options to regular expression, re2_syntax for regular expression syntax, and `re2_match` to extract matched groups.

Examples

```
panagram <- c(
  "The quick brown fox jumps over the lazy dog",
  "How vexingly quick daft zebras jump!"
)

re2_split(panagram, " quick | over | zebras ")
re2_split(panagram, " quick | over | zebras ", simplify = TRUE)

# Use compiled regexp
re <- re2_regexp("quick | over |how ", case_sensitive = FALSE)
re2_split(panagram, re)
re2_split(panagram, re, simplify = TRUE)

# Restrict number of matches
re2_split(panagram, " quick | over | zebras ", n = 2)
```

---

re2_syntax *RE2 Regular Expression Syntax*

Description

The simplest regular expression is a single literal character. Except for the metacharacters like *+?()|, characters match themselves. To match a metacharacter, escape it with a backslash: \+ matches a literal plus character.

Two regular expressions can be alternated or concatenated to form a new regular expression: if e_1 matches s and e_2 matches t, then e_1|e_2 matches s or t, and e_1e_2 matches st.

The metacharacters *, +, and ? are repetition operators: e_1* matches a sequence of zero or more (possibly different) strings, each of which match e_1; e_1+ matches one or more; e_1? matches zero or one.

The operator precedence, from weakest to strongest binding, is first alternation, then concatenation, and finally the repetition operators. Explicit parentheses can be used to force different meanings, just as in arithmetic expressions. Some examples: ab|cd is equivalent to (ab)|(cd); ab* is equivalent to a(b*).

The syntax described so far is most of the traditional Unix egrep regular expression syntax. This subset suffices to describe all regular languages: loosely speaking, a regular language is a set of strings that can be matched in a single pass through the text using only a fixed amount of memory. Newer regular expression facilities (notably Perl and those that have copied it) have added many new operators and escape sequences, which make the regular expressions more concise, and sometimes more cryptic, but usually not more powerful.

This page lists the regular expression syntax accepted by RE2. It also lists some syntax accepted by PCRE, PERL, and VIM.

kinds of single-character expressions	**examples**
any character, possibly including newline (s=true)	.
character class	[xyz]

negated character class	[^xyz]
Perl character class (see below)(link)	\d
negated Perl character class	\D
ASCII character class (see below)(link)	[[:alpha:]]
negated ASCII character class	[[:^alpha:]]
Unicode character class (one-letter name)	\pN
Unicode character class	\p{Greek}
negated Unicode character class (one-letter name)	\PN
negated Unicode character class	\P{Greek}

Composites

xy	x followed by y
x|y	x or y (prefer x)

Repetitions

x*	zero or more x, prefer more
x+	one or more x, prefer more
x?	zero or one x, prefer one
x{n,m}	n or n+1 or ... or m x, prefer more
x{n,}	n or more x, prefer more
x{n}	exactly n x
x*?	zero or more x, prefer fewer
x+?	one or more x, prefer fewer
x??	zero or one x, prefer zero
x{n,m}?	n or n+1 or ... or m x, prefer fewer
x{n,}?	n or more x, prefer fewer
x{n}?	exactly n x
x{}	(= x*) (NOT SUPPORTED) VIM
x{-}	(= x*?) (NOT SUPPORTED) VIM
x{-n}	(= x{n}?) (NOT SUPPORTED) VIM
x=	(= x?) (NOT SUPPORTED) VIM

Implementation restriction: The counting forms x{n,m}, x{n,}, and x{n} reject forms that create a minimum or maximum repetition count above 1000. Unlimited repetitions are not subject to this restriction.

Possessive repetitions

x*+	zero or more x, possessive (NOT SUPPORTED)
x++	one or more x, possessive (NOT SUPPORTED)
x?+	zero or one x, possessive (NOT SUPPORTED)
x{n,m}+	n or ... or m x, possessive (NOT SUPPORTED)
x{n,}+	n or more x, possessive (NOT SUPPORTED)
x{n}+	exactly n x, possessive (NOT SUPPORTED)

	Grouping
(re)	numbered capturing group (submatch)
(?P<name>re)	named & numbered capturing group (submatch)
(?<name>re)	named & numbered capturing group (submatch) (NOT SUPPORTED)
(?'name're)	named & numbered capturing group (submatch) (NOT SUPPORTED)
(?:re)	non-capturing group
(?flags)	set flags within current group; non-capturing
(?flags:re)	set flags during re; non-capturing
(?#text)	comment (NOT SUPPORTED)
(?|x|y|z)	branch numbering reset (NOT SUPPORTED)
(?>re)	possessive match of re (NOT SUPPORTED)
re@>	possessive match of re (NOT SUPPORTED) VIM
%(re)	non-capturing group (NOT SUPPORTED) VIM

	Flags
i	case-insensitive (default false)
m	multi-line mode: ^ and $ match begin/end line in addition to begin/end text (default false)
s	let . match \n (default false)
U	ungreedy: swap meaning of x* and x*?, x+ and x+?, etc (default false)

Flag syntax is xyz (set) or -xyz (clear) or xy-z (set xy, clear z).

	Empty strings
^	at beginning of text or line (m=true)
$	at end of text (like \z not \Z) or line (m=true)
\A	at beginning of text
\b	at ASCII word boundary (\w on one side and \W, \A, or \z on the other)
\B	not at ASCII word boundary
\g	at beginning of subtext being searched (NOT SUPPORTED) PCRE
\G	at end of last match (NOT SUPPORTED) PERL
\Z	at end of text, or before newline at end of text (NOT SUPPORTED)
\z	at end of text
(?=re)	before text matching re (NOT SUPPORTED)
(?!re)	before text not matching re (NOT SUPPORTED)
(?<=re)	after text matching re (NOT SUPPORTED)
(?<!re)	after text not matching re (NOT SUPPORTED)
re&	before text matching re (NOT SUPPORTED) VIM
re@=	before text matching re (NOT SUPPORTED) VIM
re@!	before text not matching re (NOT SUPPORTED) VIM
re@<=	after text matching re (NOT SUPPORTED) VIM
re@<!	after text not matching re (NOT SUPPORTED) VIM
\zs	sets start of match (= \K) (NOT SUPPORTED) VIM

\ze	sets end of match (NOT SUPPORTED) VIM
\%^	beginning of file (NOT SUPPORTED) VIM
\%$	end of file (NOT SUPPORTED) VIM
\%V	on screen (NOT SUPPORTED) VIM
\%#	cursor position (NOT SUPPORTED) VIM
\%'m	mark m position (NOT SUPPORTED) VIM
\%23l	in line 23 (NOT SUPPORTED) VIM
\%23c	in column 23 (NOT SUPPORTED) VIM
\%23v	in virtual column 23 (NOT SUPPORTED) VIM

Escape sequences

\a	bell (= \007)
\f	form feed (= \014)
\t	horizontal tab (= \011)
\n	newline (= \012)
\r	carriage return (= \015)
\v	vertical tab character (= \013)
*	literal *, for any punctuation character *
\123	octal character code (up to three digits)
\x7F	hex character code (exactly two digits)
\x{10FFFF}	hex character code
\C	match a single byte even in UTF-8 mode
\Q...\E	literal text ... even if ... has punctuation
\1	backreference (NOT SUPPORTED)
\b	backspace (NOT SUPPORTED) (use \010)
\cK	control char ^K (NOT SUPPORTED) (use \001 etc)
\e	escape (NOT SUPPORTED) (use \033)
\g1	backreference (NOT SUPPORTED)
\g{1}	backreference (NOT SUPPORTED)
\g{+1}	backreference (NOT SUPPORTED)
\g{-1}	backreference (NOT SUPPORTED)
\g{name}	named backreference (NOT SUPPORTED)
\g<name>	subroutine call (NOT SUPPORTED)
\g'name'	subroutine call (NOT SUPPORTED)
\k<name>	named backreference (NOT SUPPORTED)
\k'name'	named backreference (NOT SUPPORTED)
\lX	lowercase X (NOT SUPPORTED)
\ux	uppercase x (NOT SUPPORTED)
\L...\E	lowercase text ... (NOT SUPPORTED)
\K	reset beginning of $0 (NOT SUPPORTED)
\N{name}	named Unicode character (NOT SUPPORTED)
\R	line break (NOT SUPPORTED)
\U...\E	upper case text ... (NOT SUPPORTED)
\X	extended Unicode sequence (NOT SUPPORTED)
\%d123	decimal character 123 (NOT SUPPORTED) VIM
\%xFF	hex character FF (NOT SUPPORTED) VIM

\%o123	octal character 123 (NOT SUPPORTED) VIM
\%u1234	Unicode character 0x1234 (NOT SUPPORTED) VIM
\%U12345678	Unicode character 0x12345678 (NOT SUPPORTED) VIM

Character class elements

x	single character
A-Z	character range (inclusive)
\d	Perl character class
[:foo:]	ASCII character class foo
\p{Foo}	Unicode character class Foo
\pF	Unicode character class F (one-letter name)

Named character classes as character class elements

[\d]	digits (= \d)
[^\d]	not digits (= \D)
[\D]	not digits (= \D)
[^\D]	not not digits (= \d)
[[:name:]]	named ASCII class inside character class (= [:name:])
[^[:name:]]	named ASCII class inside negated character class (= [:^name:])
[\p{Name}]	named Unicode property inside character class (= \p{Name})
[^\p{Name}]	named Unicode property inside negated character class (= \P{Name})

Perl character classes (all ASCII-only)

\d	digits (= [0-9])
\D	not digits (= [^0-9])
\s	whitespace (= [\t\n\f\r])
\S	not whitespace (= [^\t\n\f\r])
\w	word characters (= [0-9A-Za-z_])
\W	not word characters (= [^0-9A-Za-z_])
\h	horizontal space (NOT SUPPORTED)
\H	not horizontal space (NOT SUPPORTED)
\v	vertical space (NOT SUPPORTED)
\V	not vertical space (NOT SUPPORTED)

ASCII character classes

[[:alnum:]]	alphanumeric (= [0-9A-Za-z])
[[:alpha:]]	alphabetic (= [A-Za-z])
[[:ascii:]]	ASCII (= [\x00-\x7F])
[[:blank:]]	blank (= [\t])
[[:cntrl:]]	control (= [\x00-\x1F\x7F])

[[:digit:]]	digits (= [0-9])
[[:graph:]]	graphical (= [!-~] = [A-Za-z0-9!"#$%&'()*+,\-./:;<=>?@[\\\]^_‘{|}~])
[[:lower:]]	lower case (= [a-z])
[[:print:]]	printable (= [-~] = [[:graph:]])
[[:punct:]]	punctuation (= [!-/:-@[-‘{-~])
[[:space:]]	whitespace (= [\t\n\v\f\r])
[[:upper:]]	upper case (= [A-Z])
[[:word:]]	word characters (= [0-9A-Za-z_])
[[:xdigit:]]	hex digit (= [0-9A-Fa-f])

Unicode character class names–general category

C	other
Cc	control
Cf	format
Cn	unassigned code points (NOT SUPPORTED)
Co	private use
Cs	surrogate
L	letter
LC	cased letter (NOT SUPPORTED)
L&	cased letter (NOT SUPPORTED)
Ll	lowercase letter
Lm	modifier letter
Lo	other letter
Lt	titlecase letter
Lu	uppercase letter
M	mark
Mc	spacing mark
Me	enclosing mark
Mn	non-spacing mark
N	number
Nd	decimal number
Nl	letter number
No	other number
P	punctuation
Pc	connector punctuation
Pd	dash punctuation
Pe	close punctuation
Pf	final punctuation
Pi	initial punctuation
Po	other punctuation
Ps	open punctuation
S	symbol
Sc	currency symbol
Sk	modifier symbol
Sm	math symbol
So	other symbol

Z	separator
Zl	line separator
Zp	paragraph separator
Zs	space separator

Unicode character class names–scripts

Adlam
Ahom
Anatolian_Hieroglyphs
Arabic
Armenian
Avestan
Balinese
Bamum
Bassa_Vah
Batak
Bengali
Bhaiksuki
Bopomofo
Brahmi
Braille
Buginese
Buhid
Canadian_Aboriginal
Carian
Caucasian_Albanian
Chakma
Cham
Cherokee
Chorasmian
Common
Coptic
Cuneiform
Cypriot
Cyrillic
Deseret
Devanagari
Dives_Akuru
Dogra
Duployan
Egyptian_Hieroglyphs
Elbasan
Elymaic
Ethiopic
Georgian
Glagolitic

Gothic
Grantha
Greek
Gujarati
Gunjala_Gondi
Gurmukhi
Han
Hangul
Hanifi_Rohingya
Hanunoo
Hatran
Hebrew
Hiragana
Imperial_Aramaic
Inherited
Inscriptional_Pahlavi
Inscriptional_Parthian
Javanese
Kaithi
Kannada
Katakana
Kayah_Li
Kharoshthi
Khitan_Small_Script
Khmer
Khojki
Khudawadi
Lao
Latin
Lepcha
Limbu
Linear_A
Linear_B
Lisu
Lycian
Lydian
Mahajani
Makasar
Malayalam
Mandaic
Manichaean
Marchen
Masaram_Gondi
Medefaidrin
Meetei_Mayek
Mende_Kikakui
Meroitic_Cursive
Meroitic_Hieroglyphs

Miao
Modi
Mongolian
Mro
Multani
Myanmar
Nabataean
Nandinagari
New_Tai_Lue
Newa
Nko
Nushu
Nyiakeng_Puachue_Hmong
Ogham
Ol_Chiki
Old_Hungarian
Old_Italic
Old_North_Arabian
Old_Permic
Old_Persian
Old_Sogdian
Old_South_Arabian
Old_Turkic
Oriya
Osage
Osmanya
Pahawh_Hmong
Palmyrene
Pau_Cin_Hau
Phags_Pa
Phoenician
Psalter_Pahlavi
Rejang
Runic
Samaritan
Saurashtra
Sharada
Shavian
Siddham
SignWriting
Sinhala
Sogdian
Sora_Sompeng
Soyombo
Sundanese
Syloti_Nagri
Syriac
Tagalog

Tagbanwa
Tai_Le
Tai_Tham
Tai_Viet
Takri
Tamil
Tangut
Telugu
Thaana
Thai
Tibetan
Tifinagh
Tirhuta
Ugaritic
Vai
Wancho
Warang_Citi
Yezidi
Yi
Zanabazar_Square

Vim character classes

\i	identifier character (NOT SUPPORTED) VIM
\I	\i except digits (NOT SUPPORTED) VIM
\k	keyword character (NOT SUPPORTED) VIM
\K	\k except digits (NOT SUPPORTED) VIM
\f	file name character (NOT SUPPORTED) VIM
\F	\f except digits (NOT SUPPORTED) VIM
\p	printable character (NOT SUPPORTED) VIM
\P	\p except digits (NOT SUPPORTED) VIM
\s	whitespace character (= [\t]) (NOT SUPPORTED) VIM
\S	non-white space character (= [^ \t]) (NOT SUPPORTED) VIM
\d	digits (= [0-9]) VIM
\D	not \d VIM
\x	hex digits (= [0-9A-Fa-f]) (NOT SUPPORTED) VIM
\X	not \x (NOT SUPPORTED) VIM
\o	octal digits (= [0-7]) (NOT SUPPORTED) VIM
\O	not \o (NOT SUPPORTED) VIM
\w	word character VIM
\W	not \w VIM
\h	head of word character (NOT SUPPORTED) VIM
\H	not \h (NOT SUPPORTED) VIM
\a	alphabetic (NOT SUPPORTED) VIM
\A	not \a (NOT SUPPORTED) VIM
\l	lowercase (NOT SUPPORTED) VIM
\L	not lowercase (NOT SUPPORTED) VIM

\u	uppercase (NOT SUPPORTED) VIM
\U	not uppercase (NOT SUPPORTED) VIM
_x	\x plus newline, for any x (NOT SUPPORTED) VIM
\c	ignore case (NOT SUPPORTED) VIM
\C	match case (NOT SUPPORTED) VIM
\m	magic (NOT SUPPORTED) VIM
\M	nomagic (NOT SUPPORTED) VIM
\v	verymagic (NOT SUPPORTED) VIM
\V	verynomagic (NOT SUPPORTED) VIM
\Z	ignore differences in Unicode combining characters (NOT SUPPORTED) VIM

	Magic
(?{code})	arbitrary Perl code (NOT SUPPORTED) PERL
(??{code})	postponed arbitrary Perl code (NOT SUPPORTED) PERL
(?n)	recursive call to regexp capturing group n (NOT SUPPORTED)
(?+n)	recursive call to relative group +n (NOT SUPPORTED)
(?-n)	recursive call to relative group -n (NOT SUPPORTED)
(?C)	PCRE callout (NOT SUPPORTED) PCRE
(?R)	recursive call to entire regexp (= (?0)) (NOT SUPPORTED)
(?&name)	recursive call to named group (NOT SUPPORTED)
(?P=name)	named backreference (NOT SUPPORTED)
(?P>name)	recursive call to named group (NOT SUPPORTED)
(?(cond)truelfalse)	conditional branch (NOT SUPPORTED)
(?(cond)true)	conditional branch (NOT SUPPORTED)
(*ACCEPT)	make regexps more like Prolog (NOT SUPPORTED)
(*COMMIT)	(NOT SUPPORTED)
(*F)	(NOT SUPPORTED)
(*FAIL)	(NOT SUPPORTED)
(*MARK)	(NOT SUPPORTED)
(*PRUNE)	(NOT SUPPORTED)
(*SKIP)	(NOT SUPPORTED)
(*THEN)	(NOT SUPPORTED)
(*ANY)	set newline convention (NOT SUPPORTED)
(*ANYCRLF)	(NOT SUPPORTED)
(*CR)	(NOT SUPPORTED)
(*CRLF)	(NOT SUPPORTED)
(*LF)	(NOT SUPPORTED)
(*BSR_ANYCRLF)	set \R convention (NOT SUPPORTED) PCRE
(*BSR_UNICODE)	(NOT SUPPORTED) PCRE

`re2_which` *Select strings that match, or find their positions*

Description

re2_subset returns strings that match a pattern. re2_which is equivalent to grep(pattern, x). It returns position of string that match a pattern. Vectorized over string and pattern. For the equivalent of grepl(pattern, x) see re2_detect.

Usage

```
re2_which(string, pattern)

re2_subset(string, pattern)
```

Arguments

string	A character vector, or an object which can be coerced to one.
pattern	Character string containing a regular expression, or a pre-compiled regular expression (or a vector of character strings and pre-compiled regular expressions). See re2_regexp for available options. See re2_syntax for regular expression syntax.

Value

re2_subset returns a character vector, and re2_which returns an integer vector.

See Also

re2_regexp for options to regular expression, re2_syntax for regular expression syntax, and re2_detect to find presence of a pattern (grep).

Examples

```
color <- c("yellowgreen", "steelblue", "GOLDENROD", "forestgreen")
re2_which(color, "o")
re2_subset(color, "o")

re2_which(c("x", "y", NA, "foo", ""), ".")
re2_subset(c("x", "y", NA, "foo", ""), ".")

# Use precompiled regexp
re <- re2_regexp("[a-z]")
re2_which(color, re)
re2_subset(color, re)

re <- re2_regexp("[a-z]", case_sensitive = FALSE)
re2_which(color, re)
re2_subset(color, re)

# Vector of patterns
re2_which(color, c("^o", "bl.e$", re, "$"))
```

Index