

Package ‘rebus.base’

July 23, 2025

Title Core Functionality for the 'rebus' Package

Version 0.0-3

Date 2017-04-25

Author Richard Cotton [aut, cre]

Maintainer Richard Cotton <richierocks@gmail.com>

Description Build regular expressions piece by piece using human readable code.
This package contains core functionality, and is primarily intended to be
used by package developers.

URL <https://github.com/richierocks/rebus.base>

BugReports <https://github.com/richierocks/rebus.base/issues>

Depends R (>= 3.1.0)

Imports stats

Suggests stringi, testthat

License Unlimited

LazyData true

RoxygenNote 6.0.1

Collate 'alternation.R' 'regex-methods.R' 'backreferences.R'
'capture.R' 'internal.R' 'grouping-and-repetition.R'
'constants.R' 'class-groups.R' 'concatenation.R'
'compound-constants.R' 'escape_special.R' 'lookaround.R'
'misc.R' 'mode-modifiers.R'

NeedsCompilation no

Repository CRAN

Date/Publication 2017-04-25 21:45:26 UTC

Contents

Anchors	2
as.regex	3

Backreferences	4
capture	5
CharacterClasses	6
char_class	9
ClassGroups	10
Concatenation	13
escape_special	14
format.regex	15
literal	15
lookahead	16
modify_mode	17
or	18
recursive	20
regex	21
repeated	21
ReplacementCase	23
SpecialCharacters	24
WordBoundaries	25
Index	27

anchors	<i>The start or end of a string.</i>
---------	--------------------------------------

Description

START matches the start of a string. END matches the end of a string. exactly makes the regular expression match the whole string, from start to end.

Usage

START

END

exactly(x)

Arguments

x A character vector.

Format

An object of class regex (inherits from character) of length 1.

Value

A character vector representing part or all of a regular expression.

Note

Caret and dollar are used as start/end delimiters, since \A and \Z are not supported by R's internal PRCE engine or stringi's ICU engine.

References

<http://www.regular-expressions.info/anchors.html> and <http://www.rexegg.com/regex-anchors.html>

See Also

[whole_word](#) and [modify_mode](#)

Examples

```
START
END

# Usage
x <- c("catfish", "tomcat", "cat")
(rx_start <- START %R% "cat")
(rx_end <- "cat" %R% END)
(rx_exact <- exactly("cat"))
stringi::stri_detect_regex(x, rx_start)
stringi::stri_detect_regex(x, rx_end)
stringi::stri_detect_regex(x, rx_exact)
```

as.regex

Convert or test for regex objects

Description

as.regex gives objects the class "regex". is.regex tests for objects of class "regex".

Usage

```
as.regex(x)
```

```
is.regex(x)
```

Arguments

x An object to test or convert.

Value

as.regex returns the inputs object, with class c("regex", "character"). is.regex returns TRUE when the input inherits from class "regex" and FALSE otherwise.

Examples

```
x <- as.regex("month.abb")
is.regex(x)
```

Backreferences	<i>Backreferences</i>
----------------	-----------------------

Description

Backreferences for replacement operations. These are used by replacement functions such as [sub](#) and [stri_replace_first_regex](#), and by the `stringi` and `stringr` match functions such as [stri_match_first_regex](#).

Usage

- REF1
- REF2
- REF3
- REF4
- REF5
- REF6
- REF7
- REF8
- REF9
- ICU_REF1
- ICU_REF2
- ICU_REF3
- ICU_REF4
- ICU_REF5
- ICU_REF6
- ICU_REF7

ICU_REF8

ICU_REF9

Format

An object of class `regex` (inherits from `character`) of length 1.

References

<http://www.regular-expressions.info/backref.html> and <http://www.rexegg.com/regex-capture.html>

See Also

[capture](#), for creating capture groups that can be referred to.

Examples

```
# For R's PCRE and Perl engines
REF1
REF2
# and so on, up to
REF9

# For stringi/stringr's ICU engine
ICU_REF1
ICU_REF2
# and so on, up to
ICU_REF9

# Usage
sub("a(b)c(d)", REF1 %R% REF2, "abcd")
stringi::stri_replace_first_regex("abcd", "a(b)c(d)", ICU_REF1 %R% ICU_REF2)
```

capture

Capture a token, or not

Description

Create a token to capture or not.

Usage

`capture(x)`

`group(x)`

`token(x)`

`engroup(x, capture)`

Arguments

`x` A character vector.

`capture` Logical If TRUE, call capture; if FALSE, call group.

Value

A character vector representing part or all of a regular expression.

References

<http://www.regular-expressions.info/brackets.html>

See Also

[or](#) for more examples

Examples

```
x <- "foo"
capture(x)
group(x)

# Usage
# capture is good with match functions
(rx_price <- capture(digit(1, Inf) %R% DOT %R% digit(2)))
(rx_quantity <- capture(digit(1, Inf)))
(rx_all <- DOLLAR %R% rx_price %R% " for " %R% rx_quantity)
stringi::stri_match_first_regex("The price was $123.99 for 12.", rx_all)

# group is mostly used with alternation. See ?or.
(rx_spread <- group("peanut butter" %|% "jam" %|% "marmalade"))
stringi::stri_extract_all_regex(
  "You can have peanut butter, jam, or marmalade on your toast.",
  rx_spread
)
```

CharacterClasses

Class Constants

Description

Match a class of values. These are typically used in combination with [char_class](#) to create new character classes.

Usage

ALPHA

ALNUM

BLANK

CNTRL

DIGIT

GRAPH

LOWER

PRINT

PUNCT

SPACE

UPPER

HEX_DIGIT

ANY_CHAR

GRAPHEME

NEWLINE

DGT

WRD

SPC

NOT_DGT

NOT_WRD

NOT_SPC

ASCII_DIGIT

ASCII_LOWER

ASCII_UPPER

ASCII_ALPHA

ASCII_ALNUM

UNMATCHABLE

Format

An object of class regex (inherits from character) of length 1.

See Also

[ClassGroups](#) for the functional form, [SpecialCharacters](#) for regex metacharacters, [Anchors](#) for constants to match the start/end of a string, [WordBoundaries](#) for constants to match the start/end of a word.

Examples

```
# R character classes
ALNUM
ALPHA
BLANK
CNTRL
DIGIT
GRAPH
LOWER
PRINT
PUNCT
SPACE
UPPER
HEX_DIGIT

# Special chars
ANY_CHAR
GRAPHEME
NEWLINE

# Generic classes
DGT
WRD
SPC

# Generic negated classes
NOT_DGT
NOT_WRD
NOT_SPC

# Non-locale-specific classes
ASCII_DIGIT
ASCII_LOWER
ASCII_UPPER
```



```

ASCII_ALPHA
ASCII_ALNUM

# An oxymoron
UNMATCHABLE

# Usage
x <- c("a1 A", "a1 a")
rx <- LOWER %R% DIGIT %R% SPACE %R% UPPER
stringi::stri_detect_regex(x, rx)

```

char_class	<i>A range or char_class of characters</i>
------------	--

Description

Group characters together in a class to match any of them (char_class) or none of them (negated_char_class).

Usage

```

char_class(...)

negated_char_class(...)

negate_and_group(...)

```

Arguments

... Character vectors.

Value

A character vector representing part or all of a regular expression.

References

<http://www.regular-expressions.info/charclass.html>

Examples

```

char_class(LOWER, ".")
negated_char_class(LOWER, ".")

# Usage
x <- (1:10) ^ 2
(rx_odd <- char_class(1, 3, 5, 7, 9))
(rx_not_odd <- negated_char_class(1, 3, 5, 7, 9))
stringi::stri_detect_regex(x, rx_odd)
stringi::stri_detect_regex(x, rx_not_odd)

```

ClassGroups

Character classes

Description

Match character classes.

Usage

```
alnum(lo, hi, char_class = TRUE)
alpha(lo, hi, char_class = TRUE)
blank(lo, hi, char_class = TRUE)
cntrl(lo, hi, char_class = TRUE)
digit(lo, hi, char_class = TRUE)
graph(lo, hi, char_class = TRUE)
lower(lo, hi, char_class = TRUE)
printable(lo, hi, char_class = TRUE)
punct(lo, hi, char_class = TRUE)
space(lo, hi, char_class = TRUE)
upper(lo, hi, char_class = TRUE)
hex_digit(lo, hi, char_class = TRUE)
any_char(lo, hi)
grapheme(lo, hi)
newline(lo, hi)
dgt(lo, hi, char_class = FALSE)
wrđ(lo, hi, char_class = FALSE)
spc(lo, hi, char_class = FALSE)
not_dgt(lo, hi, char_class = FALSE)
```

```
not_wrd(lo, hi, char_class = FALSE)
not_spc(lo, hi, char_class = FALSE)
ascii_digit(lo, hi, char_class = TRUE)
ascii_lower(lo, hi, char_class = TRUE)
ascii_upper(lo, hi, char_class = TRUE)
ascii_alpha(lo, hi, char_class = TRUE)
ascii_alnum(lo, hi, char_class = TRUE)
char_range(lo, hi, char_class = lo < hi)
```

Arguments

lo	A non-negative integer. Minimum number of repeats, when grouped.
hi	positive integer. Maximum number of repeats, when grouped.
char_class	A logical value. Should x be wrapped in a character class? If NA, the function guesses whether that's a good idea.

Value

A character vector representing part or all of a regular expression.

Note

R has many built-in locale-dependent character classes, like `[ :alnum: ]` (representing alphanumeric characters, that is lower or upper case letters or numbers). Some of these behave in unexpected ways when using the ICU engine (that is, when using `stringi` or `stringr`). See the punctuation example. For these engines, using Unicode properties ([UnicodeProperty](#)) may give you a more reliable match. There are also some generic character classes like `\w` (representing lower or upper case letters or numbers or underscores). Since version 0.0-3, these use the default `char_class = FALSE`, since they already act as character classes. Finally, there are ASCII-only ways of specifying letters like `a-zA-Z`. Which version you want depends upon how you want to deal with international characters, and the vagaries of the underlying regular expression engine. I suggest reading the [regex](#) help page and doing lots of testing.

References

<http://www.regular-expressions.info/shorthand.html> and <http://www.regexg.com/regex-quickstart.html#posix>

See Also

[regex](#), [Unicode](#)

Examples

```
# R character classes
alnum()
alpha()
blank()
cntrl()
digit()
graph()
lower()
printable()
punct()
space()
upper()
hex_digit()

# Special chars
any_char()
grapheme()
newline()

# Generic classes
dgt()
wrđ()
spc()

# Generic negated classes
not_dgt()
not_wrd()
not_spc()

# Non-locale-specific classes
ascii_digit()
ascii_lower()
ascii_upper()

# Don't provide a class wrapper
digit(char_class = FALSE) # same as DIGIT

# Match repeated values
digit(3)
digit(3, 5)
digit(0)
digit(1)
digit(0, 1)

# Ranges of characters
char_range(0, 7) # octal number

# Usage
(rx <- digit(3))
stringi::stri_detect_regex(c("123", "one23"), rx)
```

```

# Some classes behave differently under different engines
# In particular PRCE and Perl recognise all these characters
# as punctuation but ICU does not
p <- c(
  "!", "@", "#", "$", "%", "^", "&", "*", "(", ")", "[", "]", "{", "}", ";",
  ":", "'", '"', " ", "<", ">", ".", "/", "?", "\\ ", "|", "~", "~"
)
icu_matched <- stringi::stri_detect_regex(p, punct())
p[icu_matched]
p[!icu_matched]
pcre_matched <- grepl(punct(), p)
p[pcre_matched]
p[!pcre_matched]

# A grapheme is a character that can be defined by more than one code point
# PCRE does not recognise the concept.
x <- c("Chloe", "Chlo\u00e9", "Chlo\u0065\u0301")
stringi::stri_match_first_regex(x, "Chlo" %R% capture(grapheme()))

# newline() matches three types of line ending: \r, \n, \r\n.
# You can standardize line endings using
stringi::stri_replace_all_regex("foo\nbar\r\nbaz\rquux", NEWLINE, "\n")

```

Concatenation

*Combine strings together***Description**

Operator equivalent of regex.

Usage

```
x %c% y
```

```
x %R% y
```

Arguments

x A character vector.

y A character vector.

Value

A character vector representing part or all of a regular expression.

Note

%c% was the original operator for this ('c' for 'concatenate'). This is hard work to type on a QW-ERTY keyboard though, so it has been replaced with %R%.

See Also

[regex](#), [paste](#)

Examples

```
# Notice the recycling
letters %R% month.abb
```

escape_special	<i>Escape special characters</i>
----------------	----------------------------------

Description

Prefix the special characters with a backslash to make them literal characters.

Usage

```
escape_special(x, escape_brace = TRUE)
```

Arguments

- x A character vector.
- escape_brace A logical value indicating if opening braces should be escaped. If using R's internal PRCE engine or stringi's ICU engine, you want this. If using the perl engine, you don't.

Value

A character vector, with regex meta-characters escaped.

Note

Special characters inside character classes (except caret, hyphen and closing bracket in certain positions) do not need to be escaped. This function makes no attempt to parse your regular expression and decide whether or not the special character is inside a character class or not. It simply escapes every value.

Examples

```
escape_special("\\ ^ $ . | ? * + ( ) { } [ ]")
```

format.regex	<i>Print or format regex objects</i>
--------------	--------------------------------------

Description

Prints/formats objects of class regex.

Usage

```
## S3 method for class 'regex'
format(x, ...)

## S3 method for class 'regex'
print(x, encode_string = FALSE, ...)
```

Arguments

x	A regex object.
...	Passed from other format methods. Currently ignored.
encode_string	If TRUE, the regex is encoded with encodeString . This means that backslashes are doubled, compared to the default of FALSE.

Value

format.regex returns a character vector. print.regex is invoked for the side effect of printing the regex object.

Examples

```
group(1:5)
lookahead(1:5)
```

literal	<i>Treat part of a regular expression literally</i>
---------	---

Description

Treats its contents as literal characters. Equivalent to using fixed = TRUE, but for part of the pattern rather than all of it.

Usage

```
literal(x)
```

Arguments

x A character vector.

Value

A character vector representing part or all of a regular expression.

Examples

```
(rx <- digit(1, 3))
(rx_literal <- literal(rx))

# Usage
stringi::stri_detect_regex("123", rx)
stringi::stri_detect_regex("123", rx_literal)
stringi::stri_detect_regex("[[:digit:]]{1,3}", rx_literal)
```

lookahead	<i>Lookaround</i>
-----------	-------------------

Description

Zero length matching. That is, the characters are matched when detecting, but not matching or extracting.

Usage

```
lookahead(x)

negative_lookahead(x)

lookbehind(x)

negative_lookbehind(x)
```

Arguments

x A character vector.

Value

A character vector representing part or all of a regular expression.

Note

Lookbehind is not supported by R's PRCE engine. Use R's Perl engine or stringi/stringr's ICU engine.

References

<http://www.regular-expressions.info/lookaround.html> and <http://www.rexegg.com/regex-lookarounds.html>

Examples

```
x <- "foo"
lookahead(x)
negative_lookahead(x)
lookbehind(x)
negative_lookbehind(x)

# Usage
x <- c("mozambique", "qatar", "iraq")
# q followed by a character that isn't u
(rx_neg_class <- "q" %R% negated_char_class("u"))
# q not followed by a u
(rx_neg_lookahead <- "q" %R% negative_lookahead("u"))
stringi::stri_detect_regex(x, rx_neg_class)
stringi::stri_detect_regex(x, rx_neg_lookahead)
stringi::stri_extract_first_regex(x, rx_neg_class)
stringi::stri_extract_first_regex(x, rx_neg_lookahead)

# PRCE engine doesn't support lookbehind
x2 <- c("queen", "vacuum")
(rx_lookbehind <- lookbehind("q")) %R% "u"
stringi::stri_detect_regex(x2, rx_lookbehind)
try(grepl(rx_lookbehind, x2))
grepl(rx_lookbehind, x2, perl = TRUE)
```

modify_mode

Apply mode modifiers

Description

Applies one or more mode modifiers to the regular expression.

Usage

```
modify_mode(x, modes = c("i", "x", "s", "m", "J", "X"))

case_insensitive(x)

free_spacing(x)

single_line(x)

multi_line(x)
```

```
duplicate_group_names(x)
```

```
no_backslash_escaping(x)
```

Arguments

`x` A character vector.
`modes` A character vector of mode modifiers.

Value

A character vector representing part or all of a regular expression.

References

<http://www.regular-expressions.info/modifiers.html> and <http://www.rexegg.com/regex-modifiers.html>

Examples

```
x <- "foo"
case_insensitive(x)
free_spacing(x)
single_line(x)
multi_line(x)
duplicate_group_names(x)
no_backslash_escaping(x)
modify_mode(x, c("i", "J", "X"))
```

or	<i>Alternation</i>
----	--------------------

Description

Match one string or another.

Usage

```
or(..., capture = FALSE)
```

```
x %| % y
```

```
or1(x, capture = FALSE)
```

Arguments

<code>...</code>	Character vectors.
<code>capture</code>	A logical value indicating whether or not the result should be captured. See note .
<code>x</code>	A character vector.
<code>y</code>	A character vector.

Value

A character vector representing part or all of a regular expression.

Note

`or` takes multiple character vector inputs and returns a character vector of the inputs separated by pipes. `%|%` is an operator interface to this function. `or1` takes a single character vector and returns a string collapsed by pipes.

When `capture` is `TRUE`, the values are wrapped in a capture group (see [capture](#)). When `capture` is `FALSE` (the default for `or` and `or1`), the values are wrapped in a non-capture group (see [token](#)). When `capture` is `NA`, (the case for `%|%`) the values are not wrapped in anything.

References

<http://www.regular-expressions.info/alternation.html>

See Also

[paste](#)

Examples

```
# or takes an arbitrary number of arguments and groups them without capture
# Notice the recycling of inputs
or(letters, month.abb, "foo")

# or1 takes a single character vector
or1(c(letters, month.abb, "foo")) # Not the same as before!

# Capture the group
or1(letters, capture = TRUE)

# Don't create a group
or1(letters, capture = NA)

# The pipe operator doesn't group
letters %|% month.abb %|% "foo"

# Usage
(rx <- or("dog", "cat", "hippopotamus"))
stringi::stri_detect_regex(c("boondoggle", "caterwaul", "water-horse"), rx)
```

recursive	<i>Make the regular expression recursive.</i>
-----------	---

Description

Makes the regular expression (or part of it) recursive.

Usage

```
recursive(x)
```

Arguments

x	A character vector.
---	---------------------

Value

A character vector representing part or all of a regular expression.

Note

Recursion is not supported by R's internal PRCE engine or stringi's ICU engine.

References

<http://www.regular-expressions.info/recurse.html> and <http://www.rexegg.com/regex-recursion.html>

Examples

```
recursive("a")

# Recursion isn't supported by R's PRCE engine or
# stringi/stringr's ICU engine
x <- c("ab222z", "ababz", "ab", "abab")
rx <- "ab(?:R)?z"
grepl(rx, x, perl = TRUE)
try(grepl(rx, x))
try(stringi::stri_detect_regex(x, rx))
```

regex	<i>Create a regex</i>
-------	-----------------------

Description

Creates a regex object.

Usage

```
regex(...)
```

Arguments

... Passed to `paste0`.

Value

An object of class `regex`.

Note

This works like `paste0`, but the returns value has class `c("regex", "character")`.

See Also

[paste0](#) `as.regex(month.abb) regex(letters[1:5], "?")`

repeated	<i>Repeat values</i>
----------	----------------------

Description

Match repeated values.

Usage

```
repeated(x, lo, hi, lazy = FALSE, char_class = NA)
```

```
optional(x, char_class = NA)
```

```
lazy(x)
```

```
zero_or_more(x, char_class = NA)
```

```
one_or_more(x, char_class = NA)
```

Arguments

<code>x</code>	A character vector.
<code>lo</code>	A non-negative integer. Minimum number of repeats, when grouped.
<code>hi</code>	positive integer. Maximum number of repeats, when grouped.
<code>lazy</code>	A logical value. Should repetition be matched lazily or greedily?
<code>char_class</code>	A logical value. Should <code>x</code> be wrapped in a character class? If NA, the function guesses whether that's a good idea.

Value

A character vector representing part or all of a regular expression.

References

<http://www.regular-expressions.info/repeat.html> and <http://www.rexegg.com/regex-quantifiers.html>

Examples

```
# Can match constants or class values
repeated(GRAPH, 2, 5)
repeated(graph(), 2, 5) # same

# Short cuts for special cases
optional(blank()) # same as repeated(blank(), 0, 1)
zero_or_more(hex_digit()) # same as repeated(hex_digit(), 0, Inf)
one_or_more(printable()) # same as repeated(printable(), 1, Inf)

# 'Lazy' matching (match smallest no. of chars)
repeated(cntrl(), 2, 5, lazy = TRUE)
lazy(one_or_more(cntrl()))

# Overriding character class wrapping
repeated(ANY_CHAR, 2, 5, char_class = FALSE)

# Usage
x <- "1234567890"
stringi::stri_extract_first_regex(x, one_or_more(DIGIT))
stringi::stri_extract_first_regex(x, repeated(DIGIT, lo = 3, hi = 6))
stringi::stri_extract_first_regex(x, lazy(repeated(DIGIT, lo = 3, hi = 6)))

col <- c("color", "colour")
stringi::stri_detect_regex(col, "colo" %R% optional("u") %R% "r")
```

ReplacementCase	<i>Force the case of replacement values</i>
-----------------	---

Description

Forces replacement values to be upper or lower case. Only supported by Perl regular expressions.

Usage

`as_lower(x)`

`as_upper(x)`

Arguments

`x` A character vector.

Value

A character vector representing part or all of a regular expression.

References

<http://www.regular-expressions.info/replacecase.html>

Examples

```
# Convert to title case using Perl regex
x <- "In caSE of DISASTER, PuLL tHe CoRd"
matching_rx <- capture(WRD) %R% capture(wrd(1, Inf))
replacement_rx <- as_upper(REF1) %R% as_lower(REF2)
gsub(matching_rx, replacement_rx, x, perl = TRUE)

# PCRE and ICU do not currently support this operation
# The next lines are intended to return gibberish
gsub(matching_rx, replacement_rx, x)
replacement_rx_icu <- as_upper(ICU_REF1) %R% as_lower(ICU_REF2)
stringi::stri_replace_all_regex(x, matching_rx, replacement_rx_icu)
```

SpecialCharacters	<i>Special characters</i>
-------------------	---------------------------

Description

Constants to match special characters.

Usage

BACKSLASH

CARET

DOLLAR

DOT

PIPE

QUESTION

STAR

PLUS

OPEN_PAREN

CLOSE_PAREN

OPEN_BRACKET

CLOSE_BRACKET

OPEN_BRACE

Format

An object of class `regex` (inherits from `character`) of length 1.

References

<http://www.regular-expressions.info/characters.html>

See Also

[escape_special](#) for the functional form, [CharacterClasses](#) for regex metacharacters, [Anchors](#) for constants to match the start/end of a string, [WordBoundaries](#) for constants to match the start/end of a word.

Examples

```

BACKSLASH
CARET
DOLLAR
DOT
PIPE
QUESTION
STAR
PLUS
OPEN_PAREN
CLOSE_PAREN
OPEN_BRACKET
CLOSE_BRACKET
OPEN_BRACE

# Usage
x <- "\\^$."
rx <- BACKSLASH %R% CARET %R% DOLLAR %R% DOT
stringi::stri_detect_regex(x, rx)
# No escapes - these chars have special meaning inside regex
stringi::stri_detect_regex(x, x)

# Usually closing brackets can be matched without escaping
stringi::stri_detect_regex("]", "]")
# If you want to match a closing bracket inside a character class
# the closing bracket must be placed first
(rx <- char_class("[a]"))
stringi::stri_detect_regex("]", rx)
# ICU and Perl also allows you to place the closing bracket in
# other positions if you escape it
(rx <- char_class("a", CLOSE_BRACKET))
stringi::stri_detect_regex("]", rx)
grepl(rx, "]", perl = TRUE)
# PCRE does not allow this
grepl(rx, "]")

```

WordBoundaries

Word boundaries

Description

BOUNDARY matches a word boundary. `whole_word` wraps a regex in word boundary tokens to match a whole word.

Usage

BOUNDARY

NOT_BOUNDARY

`whole_word(x)`

Arguments

x A character vector.

Format

An object of class `regex` (inherits from `character`) of length 1.

Value

A character vector representing part or all of a regular expression.

References

<http://www.regular-expressions.info/wordboundaries.html> and <http://www.rexegg.com/regex-boundaries.html>

See Also

[ALPHA](#), [BACKSLASH](#), [START](#)

Examples

```
BOUNDARY
NOT_BOUNDARY

# Usage
x <- c("the catfish miaowed", "the tomcat miaowed", "the cat miaowed")
(rx_before <- BOUNDARY %R% "cat")
(rx_after <- "cat" %R% BOUNDARY)
(rx_whole_word <- whole_word("cat"))
stringi::stri_detect_regex(x, rx_before)
stringi::stri_detect_regex(x, rx_after)
stringi::stri_detect_regex(x, rx_whole_word)
```

Index

* datasets

- Anchors, [2](#)
- Backreferences, [4](#)
- CharacterClasses, [6](#)
- SpecialCharacters, [24](#)
- WordBoundaries, [25](#)
- %R% (Concatenation), [13](#)
- %c% (Concatenation), [13](#)
- ALNUM (CharacterClasses), [6](#)
- alnum (ClassGroups), [10](#)
- ALPHA, [26](#)
- ALPHA (CharacterClasses), [6](#)
- alpha (ClassGroups), [10](#)
- Anchors, [2](#), [8](#), [24](#)
- ANY_CHAR (CharacterClasses), [6](#)
- any_char (ClassGroups), [10](#)
- as.regex, [3](#)
- as_lower (ReplacementCase), [23](#)
- as_upper (ReplacementCase), [23](#)
- ASCII_ALNUM (CharacterClasses), [6](#)
- ascii_alnum (ClassGroups), [10](#)
- ASCII_ALPHA (CharacterClasses), [6](#)
- ascii_alpha (ClassGroups), [10](#)
- ASCII_DIGIT (CharacterClasses), [6](#)
- ascii_digit (ClassGroups), [10](#)
- ASCII_LOWER (CharacterClasses), [6](#)
- ascii_lower (ClassGroups), [10](#)
- ASCII_UPPER (CharacterClasses), [6](#)
- ascii_upper (ClassGroups), [10](#)
- Backreferences, [4](#)
- BACKSLASH, [26](#)
- BACKSLASH (SpecialCharacters), [24](#)
- BLANK (CharacterClasses), [6](#)
- blank (ClassGroups), [10](#)
- BOUNDARY (WordBoundaries), [25](#)
- capture, [5](#), [5](#), [19](#)
- CARET (SpecialCharacters), [24](#)

- case_insensitive (modify_mode), [17](#)
- char_class, [6](#), [9](#)
- char_range (ClassGroups), [10](#)
- CharacterClasses, [6](#), [24](#)
- ClassGroups, [8](#), [10](#)
- CLOSE_BRACKET (SpecialCharacters), [24](#)
- CLOSE_PAREN (SpecialCharacters), [24](#)
- CNTRL (CharacterClasses), [6](#)
- cntrl (ClassGroups), [10](#)
- Concatenation, [13](#)
- DGT (CharacterClasses), [6](#)
- dgt (ClassGroups), [10](#)
- DIGIT (CharacterClasses), [6](#)
- digit (ClassGroups), [10](#)
- DOLLAR (SpecialCharacters), [24](#)
- DOT (SpecialCharacters), [24](#)
- duplicate_group_names (modify_mode), [17](#)
- encodeString, [15](#)
- END (Anchors), [2](#)
- engroup (capture), [5](#)
- escape_special, [14](#), [24](#)
- exactly (Anchors), [2](#)
- format.regex, [15](#)
- free_spacing (modify_mode), [17](#)
- GRAPH (CharacterClasses), [6](#)
- graph (ClassGroups), [10](#)
- GRAPHEME (CharacterClasses), [6](#)
- grapheme (ClassGroups), [10](#)
- group (capture), [5](#)
- HEX_DIGIT (CharacterClasses), [6](#)
- hex_digit (ClassGroups), [10](#)
- ICU_REF1 (Backreferences), [4](#)
- ICU_REF2 (Backreferences), [4](#)
- ICU_REF3 (Backreferences), [4](#)
- ICU_REF4 (Backreferences), [4](#)

- ICU_REF5 (Backreferences), 4
- ICU_REF6 (Backreferences), 4
- ICU_REF7 (Backreferences), 4
- ICU_REF8 (Backreferences), 4
- ICU_REF9 (Backreferences), 4
- is.regex (as.regex), 3

- lazy (repeated), 21
- literal, 15
- lookahead, 16
- lookbehind (lookahead), 16
- LOWER (CharacterClasses), 6
- lower (ClassGroups), 10

- modify_mode, 3, 17
- multi_line (modify_mode), 17

- negate_and_group (char_class), 9
- negated_char_class (char_class), 9
- negative_lookahead (lookahead), 16
- negative_lookbehind (lookahead), 16
- NEWLINE (CharacterClasses), 6
- newline (ClassGroups), 10
- no_backslash_escaping (modify_mode), 17
- NOT_BOUNDARY (WordBoundaries), 25
- NOT_DGT (CharacterClasses), 6
- not_dgt (ClassGroups), 10
- NOT_SPC (CharacterClasses), 6
- not_spc (ClassGroups), 10
- NOT_WRD (CharacterClasses), 6
- not_wrd (ClassGroups), 10

- one_or_more (repeated), 21
- OPEN_BRACE (SpecialCharacters), 24
- OPEN_BRACKET (SpecialCharacters), 24
- OPEN_PAREN (SpecialCharacters), 24
- optional (repeated), 21
- or, 6, 18
- or1 (or), 18

- paste, 14, 19
- paste0, 21
- PIPE (SpecialCharacters), 24
- PLUS (SpecialCharacters), 24
- PRINT (CharacterClasses), 6
- print.regex (format.regex), 15
- printable (ClassGroups), 10
- PUNCT (CharacterClasses), 6
- punct (ClassGroups), 10

- QUESTION (SpecialCharacters), 24

- recursive, 20
- REF1 (Backreferences), 4
- REF2 (Backreferences), 4
- REF3 (Backreferences), 4
- REF4 (Backreferences), 4
- REF5 (Backreferences), 4
- REF6 (Backreferences), 4
- REF7 (Backreferences), 4
- REF8 (Backreferences), 4
- REF9 (Backreferences), 4
- regex, 11, 14, 21
- repeated, 21
- ReplacementCase, 23

- single_line (modify_mode), 17
- SPACE (CharacterClasses), 6
- space (ClassGroups), 10
- SPC (CharacterClasses), 6
- spc (ClassGroups), 10
- SpecialCharacters, 8, 24
- STAR (SpecialCharacters), 24
- START, 26
- START (Anchors), 2
- stri_match_first_regex, 4
- stri_replace_first_regex, 4
- sub, 4

- token, 19
- token (capture), 5

- Unicode, 11
- UnicodeProperty, 11
- UNMATCHABLE (CharacterClasses), 6
- UPPER (CharacterClasses), 6
- upper (ClassGroups), 10

- whole_word, 3
- whole_word (WordBoundaries), 25
- WordBoundaries, 8, 24, 25
- WRD (CharacterClasses), 6
- wrd (ClassGroups), 10

- zero_or_more (repeated), 21