

Package: RVerbalExpressions (via r-universe)

September 13, 2024

Title Create Regular Expressions Easily

Version 0.1.1

Description Build regular expressions using grammar and functionality inspired by <<https://github.com/VerbalExpressions>>. Usage of the %>% is encouraged to build expressions in a chain-like fashion.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.2.3

Depends R (>= 2.10)

Imports magrittr

Suggests testthat, knitr, rmarkdown, covr, stringr

VignetteBuilder knitr

URL <https://github.com/VerbalExpressions/RVerbalExpressions>

BugReports <https://github.com/VerbalExpressions/RVerbalExpressions/issues>

Repository <https://verbalexpressions.r-universe.dev>

RemoteUrl <https://github.com/verbalexpressions/rverbalexpressions>

RemoteRef HEAD

RemoteSha 23b940a21c554944a3ba1feedb401e85749446c2

Contents

rx	2
rx_alnum	3
rx_alpha	4
rx_anything	4
rx_anything_but	5
rx_any_of	6
rx_avoid_prefix	7
rx_begin_capture	8

rx_digit	8
rx_either_of	9
rx_end_capture	10
rx_end_of_line	10
rx_find	11
rx_line_break	12
rx_lowercase	12
rx_maybe	13
rx_multiple	14
rx_none_or_more	14
rx_not	15
rx_one_or_more	16
rx_punctuation	17
rx_range	17
rx_seek_prefix	18
rx_something	19
rx_something_but	19
rx_space	20
rx_start_of_line	21
rx_tab	22
rx_uppercase	22
rx_whitespace	23
rx_with_any_case	24
rx_word	25
rx_word_char	26
rx_word_edge	26
sanitize	27

Index **28**

rx	<i>Constructs a Verbal Expression</i>
----	---------------------------------------

Description

Add this to the beginning of every verbal expression chain. This simply returns an empty character vector so that the next step in the chain can provide a value without explicitly writing value = "blah".

Usage

rx()

Examples

```
rx()

# this
rx() %>%
  rx_find("cat") %>%
  rx_anything() %>%
  rx_find("dog")

# instead of
rx_find(value = "cat") %>%
  rx_anything() %>%
  rx_find("dog")
```

rx_alnum	<i>Match alphanumeric characters.</i>
----------	---------------------------------------

Description

Matches both letters (case insensitive) and numbers (a through z and 0 through 9).

Usage

```
rx_alnum(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match alphanumeric characters). Use TRUE to <i>not</i> match alphanumeric characters.

Examples

```
rx_alnum()
rx_alnum(inverse = TRUE)

# create an expression
x <- rx_alnum()

# create input
string <- "Apple 1!"

# extract match
regmatches(string, gregexpr(x, string))
```

rx_alpha	<i>Match alphabetic characters.</i>
----------	-------------------------------------

Description

Matches letters (case insensitive) only.

Usage

```
rx_alpha(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match alphabetic characters). Use TRUE to <i>not</i> match alphabetic characters.

Examples

```
rx_alpha()
rx_alpha(inverse = TRUE)

# create an expression
x <- rx_alpha()

# create input
string <- "Apple 1!"

# extract match
regmatches(string, gregexpr(x, string))
```

rx_anything	<i>Match any character(s) any (including zero) number of times.</i>
-------------	---

Description

This expression will match everything except line breaks using the *dot* and the *star*. The Dot `.` is a *metacharacter* and the Star `*` is a *quantifier*. When combined the expression is considered greedy because it will match everything (except line breaks) 0 or more times.

Usage

```
rx_anything(.data = NULL, mode = "greedy")
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
mode	Matching mode (greedy (default) or lazy). Lazy matching stops after the first match, greedy continues searching until end of the string and then back-tracks to the last match.

References

Dot: <https://www.regular-expressions.info/dot.html>

Star Quantifier: <https://www.regular-expressions.info/repeat.html>

Greedy and Lazy Quantifiers: <https://www.regular-expressions.info/repeat.html#greedy>

Examples

```
rx_anything()
rx_anything(mode = "lazy")

x <- rx() %>%
  rx_start_of_line() %>%
  rx_anything() %>%
  rx_end_of_line()

grepl(x, "anything!") # this should be true
grepl(rx_anything(), "") # this should be true
grepl(rx_something(), "") # this should be false
```

rx_anything_but	<i>Match any character(s) except these any (including zero) number of times.</i>
-----------------	--

Description

This expression will match everything except whatever characters the user specifies in the value parameter. It does this by adding a caret symbol ^ at the beginning of a character set []. Typing a caret after the opening square bracket negates the character class. The result is that the character class matches any character that is not in the character class. Unlike the dot, negated character classes also match (invisible) line break characters. If you don't want a negated character class to match line breaks, you need to include the line break characters in the class.

Usage

```
rx_anything_but(.data = NULL, value, mode = "greedy")
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Characters to not match
mode	Matching mode (greedy (default) or lazy). Lazy matching stops after the first match, greedy continues searching until end of the string and then back-tracks to the last match.

References

Character Class: <https://www.regular-expressions.info/charclass.html>

Examples

```
rx_anything_but(value = "abc")
```

rx_any_of	<i>Match any of these characters exactly once.</i>
-----------	--

Description

Constructs a *character class*, sometimes called a *character set*. With this particular expression, you can tell the regex engine to match only one out of several characters. It does this by simply placing the characters you want to match between square brackets.

Usage

```
rx_any_of(.data = NULL, value)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Expression to optionally match

References

Character class: <https://www.regular-expressions.info/charclass.html>

Examples

```
rx_any_of(value = "abc")

# create an expression
x <- rx_any_of(value = "abc")

grepl(x, "c") # should be true
grepl(x, "d") # should be false
```

```

y <- rx() %>%
  rx_find("gr") %>%
  rx_any_of("ae") %>%
  rx_find("y")

regmatches("gray", regexec(y, "gray"))[[1]]
regmatches("grey", regexec(y, "grey"))[[1]]

```

rx_avoid_prefix	<i>Negative lookahead functions</i>
-----------------	-------------------------------------

Description

This function facilitates matching by providing negative assurances for surrounding symbols/groups of symbols. It allows for building expressions that are dependent on context of occurrence.

Usage

```
rx_avoid_prefix(.data = NULL, value)
```

```
rx_avoid_suffix(.data = NULL, value)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Exact expression to match

Examples

```

# matches any number of digits, but not preceded by "USD"
rx() %>%
  rx_avoid_prefix('USD') %>%
  rx_digit() %>%
  rx_one_or_more()

#matches a digit, but not followed by " dollars"
rx() %>%
  rx_digit() %>%
  rx_avoid_suffix(' dollars')

```

rx_begin_capture	<i>Begin a capture group.</i>
------------------	-------------------------------

Description

Begin a capture group.

Usage

```
rx_begin_capture(.data = NULL)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
-------	--

Details

Capture groups are used to extract data from within the regular expression match for further processing.

rx_digit	<i>Match a digit (0–9).</i>
----------	-----------------------------

Description

The function `rx_digit()` looks for tabs with the following expression: `%d` and matches single digit. Plural version matches specified number of digits `n` (equivalent to `rx_digit() %>% rx_count(n)`).

Usage

```
rx_digit(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match digit characters). Use TRUE to <i>not</i> match digit characters.

Examples

```

rx_digit()
rx_digit(inverse = TRUE)

# create an expression
x <- rx_digit()

# create input
string <- "1 apple"

# extract match
regmatches(string, regexpr(x, string))

```

rx_either_of	<i>Alternatively, match either expression.</i>
--------------	--

Description

Expression to match instead. If both expressions exists, both will be returned. This just adds the vertical bar | often called an *alternator* which allows the user to find this *or* that, or both!

Usage

```
rx_either_of(.data, ...)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
...	A character vector

Examples

```

x <- rx() %>%
  rx_either_of("cat", "dog") %>%
  rx_space() %>%
  rx_find("food")

string <- c("dog food", "cat food", "fish food")

grep(x, string, value = TRUE)

```

rx_end_capture	<i>End a capture group.</i>
----------------	-----------------------------

Description

End a capture group.

Usage

```
rx_end_capture(.data = NULL)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
-------	--

Details

Capture groups are used to extract data from within the regular expression match for further processing.

rx_end_of_line	<i>Match the expression only if it appears till the end of the line.</i>
----------------	--

Description

Control whether to match the expression only if it appears till the end of the line. Basically, append a \$ to the end of the expression. The dollar sign is considered an *anchor* and matches the position of characters. It can be used to "anchor" the regex match at a certain position, in this case the dollar sign matches right after the last character in the string.

Usage

```
rx_end_of_line(.data = NULL, enable = TRUE)
```

Arguments

.data	Expression to match, typically pulled from the pipe %>%
enable	Whether to enable this behavior, defaults to TRUE

References

Anchors: <https://www.regular-expressions.info/anchors.html>

Examples

```

rx_end_of_line(enable = TRUE)
rx_end_of_line(enable = FALSE)
rx_end_of_line("abc", enable = TRUE)

# create expression
x <- rx() %>%
  rx_start_of_line(FALSE) %>%
  rx_find("apple") %>%
  rx_end_of_line()

grepl(x, "apples") # should be false
grepl(x, "apple") # should be true

```

rx_find	<i>Match an expression.</i>
---------	-----------------------------

Description

Identify a specific pattern exactly.

Usage

```
rx_find(.data = NULL, value)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Exact expression to match

References

Capturing group: <https://www.regular-expressions.info/brackets.html>
 Stack Overflow: <https://stackoverflow.com/questions/3512471>

Examples

```

rx_find(value = "apple")

# create expression
x <- rx_find(value = "apples")

grepl(x, "apple") # should be false
grepl(x, "apples") # should be true

```

rx_line_break	<i>Match a line break.</i>
---------------	----------------------------

Description

This expression looks for line breaks, both Unix and Windows style by using the appropriate *non printable characters*.

Usage

```
rx_line_break(.data = NULL)
```

Arguments

.data Expression to append, typically pulled from the pipe %>%

References

Unix style: <https://codepoints.net/U+000A>

Windows style: <https://codepoints.net/U+000D>

Non printable character: <https://www.regular-expressions.info/nonprint.html>

Examples

```
rx_line_break()

# create an expression
x <- rx_line_break()

# create input
string <- "foo\nbar"

# extract match
regmatches(string, regexpr(x, string))
```

rx_lowercase	<i>Match lower case letters.</i>
--------------	----------------------------------

Description

Matches lower case letters only.

Usage

```
rx_lowercase(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match lower case). Use TRUE to <i>not</i> match lower case.

Examples

```
rx_lowercase()
rx_lowercase(inverse = TRUE)

# create an expression
x <- rx_lowercase()
y <- rx_lowercase(inverse = TRUE)

# create input
string <- "Apple 1!"

# extract match
regmatches(string, gregexpr(x, string))
regmatches(string, gregexpr(y, string))
```

rx_maybe	<i>Optionally match an expression.</i>
----------	--

Description

This expression uses a *quantifier* ? to optionally match things. Specifically, the question mark makes the preceding token in the regular expression optional.

Usage

```
rx_maybe(.data = NULL, value)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Expression to optionally match

References

Quantifiers: <https://www.regular-expressions.info/optional.html>

Examples

```

rx_maybe(value = "abc")

# create expression
x <- rx() %>%
  rx_start_of_line() %>%
  rx_maybe("abc") %>%
  rx_end_of_line(enable = FALSE)

grepl(x, "xyz") # should be true

```

rx_multiple	<i>Match the previous group any number of times.</i>
-------------	--

Description

Match the previous group any number of times.

Usage

```
rx_multiple(.data = NULL, value = NULL, min = NULL, max = NULL)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Item to match
min	Minimum number of times it should be present
max	Maximum number of times it should be present

rx_none_or_more	<i>Match the previous stuff zero or many times.</i>
-----------------	---

Description

This function simply adds a * to the end of the expression.

Usage

```
rx_none_or_more(.data = NULL, mode = "greedy")
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
mode	Matching mode (greedy (default) or lazy). Lazy matching stops after the first match, greedy continues searching until end of the string and then back-tracks to the last match.

Examples

```

rx_none_or_more()

# create an expression
x <- rx() %>%
  rx_find("a") %>%
  rx_none_or_more()

# create input
input <- "aaa"

# extract match
regmatches(input, regexpr(x, input))

```

rx_not

Ensure that the parameter does not follow.

Description

This expression uses a *negative lookahead* to ensure the value given does not follow the previous verbal expression, `perl = TRUE` is required. For example, if you were to look for the letter *q* but not the letter *u* you might translate this to, "find the letter *q* everytime the letter *u* does *not* come after it".

Usage

```
rx_not(.data = NULL, value)
```

Arguments

<code>.data</code>	Expression to append, typically pulled from the pipe <code>%>%</code>
<code>value</code>	Value to ensure absence of

References

Negative lookahead: <https://www.regular-expressions.info/lookaround.html>

Examples

```

rx_not(value = "FEB-28")

# construct expression
x <- rx() %>%
  rx_start_of_line() %>%
  rx_find('FEB-29') %>%
  rx_not("FEB-28")

# create a string
string <- c("FEB-29-2017", "FEB-28-2017")

```

```
# extract matches, perl = TRUE is required for negative lookahead
regmatches(string, regexpr(x, string, perl = TRUE))

# another example
rx() %>%
  rx_find("q") %>%
  rx_not("u") %>%
  grepl(x = c("qu", "qa", "qq", "q", "q u"), perl = TRUE)
```

 rx_one_or_more

Match the previous stuff one or more times.

Description

This function simply adds a + to the end of the expression.

Usage

```
rx_one_or_more(.data = NULL, mode = "greedy")
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
mode	Matching mode (greedy (default) or lazy). Lazy matching stops after the first match, greedy continues searching until end of the string and then back-tracks to the last match.

Examples

```
rx_one_or_more()

# create an expression
x <- rx() %>%
  rx_find("a") %>%
  rx_one_or_more()

# create input
input <- "aaa"

# extract match
regmatches(input, regexpr(x, input))
```

rx_punctuation	<i>Match punctuation characters.</i>
----------------	--------------------------------------

Description

Matches punctuation characters only: !\"#\$%&'()*+,-./:;<=>?@[\\]^_`{|}~.

Usage

```
rx_punctuation(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match punctuation). Use TRUE to <i>not</i> match punctuation.

Examples

```
rx_punctuation()
rx_punctuation(inverse = TRUE)

# create an expression
x <- rx_punctuation()

# create input
string <- 'Apple !!'

# extract match
regmatches(string, gregexpr(x, string))

# dont extract punctuation
y <- rx_punctuation(inverse = TRUE)
regmatches(string, gregexpr(y, string))
```

rx_range	<i>Match any character within the range defined by the parameters.</i>
----------	--

Description

Value parameter will be interpreted as pairs. For example, `range(c('a', 'z', '0', '9'))` will be interpreted to mean any character within the ranges a–z (ascii x–y) or 0–9 (ascii x–y). The method expects an even number of parameters; unpaired parameters are ignored.

Usage

```
rx_range(.data = NULL, value)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Range of characters. The method expects an even number of parameters; unpaired parameters are ignored.

Examples

```
rx_range(value = c('1', '3'))

# create an expression
x <- rx_range(value = c('1', '3'))

grepl(x, "2") # should be true
grepl(x, "4") # should be false
```

rx_seek_prefix	<i>Positive lookahead functions</i>
----------------	-------------------------------------

Description

This function facilitates matching by providing assurances for surrounding symbols/groups of symbols. It allows for building expressions that are dependent on context of occurrence.

Usage

```
rx_seek_prefix(.data = NULL, value)
```

```
rx_seek_suffix(.data = NULL, value)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Exact expression to match

Examples

```
# this will match anything between square brackets
rx() %>%
  rx_seek_prefix("[") %>%
  rx_anything("lazy") %>%
  rx_seek_suffix("]")
```

rx_something	<i>Match any character(s) at least once.</i>
--------------	--

Description

This expression is almost identical to `rx_anything()` with one major exception, a `+` is used instead of a `*`. This means `rx_something()` expects *something* whereas `anything()` expects *anything* including... nothing!

Usage

```
rx_something(.data = NULL, mode = "greedy")
```

Arguments

<code>.data</code>	Expression to append, typically pulled from the pipe <code>%>%</code>
<code>mode</code>	Matching mode (greedy (default) or lazy). Lazy matching stops after the first match, greedy continues searching until end of the string and then back-tracks to the last match.

References

Metacharacters: <https://www.regular-expressions.info/characters.html#special>

Greedy and Lazy Quantifiers: <https://www.regular-expressions.info/repeat.html#greedy>

Examples

```
rx_something()

# construct an expression
x <- rx_something()

grepl(x, "something!") # this should be true
grepl(x, "")           # this should be false
grepl(rx_anything(), "") # this should be true
```

rx_something_but	<i>Match any character(s) except these at least once.</i>
------------------	---

Description

This expression is almost identical to `rx_anything_but()` with one major exception, a `+` is used instead of a `*`. This means `rx_something_but()` expects *something* whereas `rx_anything_but()` expects *anything* including... nothing!

Usage

```
rx_something_but(.data = NULL, value, mode = "greedy")
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
value	Expression to optionally match
mode	Matching mode (greedy (default) or lazy). Lazy matching stops after the first match, greedy continues searching until end of the string and then back-tracks to the last match.

References

Metacharacters: <https://www.regular-expressions.info/characters.html#special>

Greedy and Lazy Quantifiers: <https://www.regular-expressions.info/repeat.html#greedy>

Examples

```
rx_something_but(value = "abc")

# create an expression
x <- rx_something_but(value = "python")

grepl(x, "R") # should be true
grepl(x, "py") # should be false
```

rx_space	<i>Match a space character.</i>
----------	---------------------------------

Description

Matches a space character.

Usage

```
rx_space(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match space). Use TRUE to <i>not</i> match space.

Examples

```
# match space, default
rx_space()

# dont match space
rx_space(inverse = TRUE)

# create an expression
x <- rx_space()

# create input
string <- "1 apple\t"

# extract match
regmatches(string, regexpr(x, string))

# extract no whitespace by inverting behavior
y <- rx_space(inverse = TRUE)
regmatches(string, gregexpr(y, string))
```

rx_start_of_line	<i>Match the expression only if it appears from beginning of line.</i>
------------------	--

Description

Control whether to match the expression only if it appears from the beginning of the line.

Usage

```
rx_start_of_line(.data = NULL, enable = TRUE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
enable	Whether to enable this behavior, defaults to TRUE

Examples

```
rx_start_of_line(enable = TRUE)
rx_start_of_line(enable = FALSE)

# create expression
x <- rx() %>%
  rx_start_of_line() %>%
  rx_find("apple")

grepl(x, "pineapple") # should be false
grepl(x, "apple")    # should be true
```

rx_tab	<i>Match a tab character.</i>
--------	-------------------------------

Description

Match a tab character.

Usage

```
rx_tab(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match tabs). Use TRUE to <i>not</i> match tabs.

Details

This function is looks for tabs with the following expression: `\t`

1. Tab character: <https://codepoints.net/U+0009>

Examples

```
rx_tab()
rx_tab(inverse = TRUE)

# create an expression
x <- rx_tab()

# create input
string <- "foo\tbar"

# extract match
regmatches(string, regexpr(x, string))
```

rx_uppercase	<i>Match upper case letters.</i>
--------------	----------------------------------

Description

Matches upper case letters only.

Usage

```
rx_uppercase(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match upper case). Use TRUE to <i>not</i> match upper case.

Examples

```
rx_uppercase()
rx_uppercase(inverse = TRUE)

# create an expression
x <- rx_uppercase()
y <- rx_uppercase(inverse = TRUE)

# create input
string <- "Apple 1!"

# extract match
regmatches(string, gregexpr(x, string))
regmatches(string, gregexpr(y, string))
```

rx_whitespace	<i>Match a whitespace character.</i>
---------------	--------------------------------------

Description

Match a whitespace character.

Usage

```
rx_whitespace(.data = NULL, inverse = FALSE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
inverse	Invert match behavior, defaults to FALSE (match whitespace). Use TRUE to <i>not</i> match whitespace.

Details

Match a whitespace character (one of space, tab, carriage return, new line, vertical tab and form feed).

1. space: <https://codepoints.net/U+0020>
2. tab: <https://codepoints.net/U+0009>
3. carriage return: <https://codepoints.net/U+000D>
4. new line: <https://codepoints.net/U+000A>
5. vertical tab: <https://codepoints.net/U+000B>
6. form feed: <https://codepoints.net/U+000C>

Examples

```
# match whitespace, default
rx_whitespace()

# dont match whitespace
rx_whitespace(inverse = TRUE)

# create an expression
x <- rx_whitespace()

# create input
string <- "1 apple"

# extract match
regmatches(string, regexpr(x, string))

# extract no whitespace by inverting behavior
y <- rx_whitespace(inverse = TRUE)
regmatches(string, gregexpr(y, string))
```

rx_with_any_case *Control case-insensitive matching.*

Description

Control case-insensitive matching.

Usage

```
rx_with_any_case(.data = NULL, enable = TRUE)
```

Arguments

.data	Expression to append, typically pulled from the pipe %>%
enable	Whether to enable this behavior

Details

Equivalent to adding or removing the *i* modifier.

Examples

```
rx_with_any_case()

# case insensitive
x <- rx() %>%
  rx_find("abc") %>%
  rx_with_any_case()
```



```
# case sensitive
y <- rx() %>%
  rx_find("abc") %>%
  rx_with_any_case(enable = FALSE)

grepl(x, "ABC") # should be true
grepl(y, "ABC") # should be false
```

rx_word

Match a word.

Description

Match a word—a string of word characters (a–z, A–Z, 0–9 or `_`). This function looks for tabs with the following expression: `\w+`

Usage

```
rx_word(.data = NULL)
```

Arguments

`.data` Expression to append, typically pulled from the pipe `%>%`

Examples

```
rx_word()

# create an expression
x <- rx_word()

# create inputs
string1 <- "foo_bar"
string2 <- "foo-bar"

# extract matches
regmatches(string1, regexpr(x, string1))
regmatches(string2, regexpr(x, string2)) # doesn't match -
```

rx_word_char	<i>Match a word character.</i>
--------------	--------------------------------

Description

Match a word character (a–z, A–Z, 0–9 or _).

Usage

```
rx_word_char(.data = NULL)
```

Arguments

.data Expression to append, typically pulled from the pipe %>%

Examples

```
rx_word_char()
# Same as rx_word()
x <- rx_word_char() %>%
  rx_one_or_more()
```

rx_word_edge	<i>Find beginning or end of a word.</i>
--------------	---

Description

Match beginning or end of a word—a string consisting of of word characters (a–z, A–Z, 0–9 or _).

Usage

```
rx_word_edge(.data = NULL)
```

Arguments

.data Expression to append, typically pulled from the pipe %>%

Examples

```
rx_word_edge()

x <- rx() %>%
  rx_word_edge() %>%
  rx_alpha() %>%
  rx_one_or_more() %>%
  rx_word_edge()

# create inputs
string1 <- "foobar"
string2 <- "foo 23a bar"

# matches 'foobar'
regmatches(string1, regexpr(x, string1))
# matches 'foo' and 'bar' separately
regmatches(string2, gregexpr(x, string2))
```

sanitize

Escape characters expected special by regex engines

Description

Takes a string and escapes all characters considered special by the regex engine. This is used internally when you add a string to the `value` parameter in most of the available functions. It is exported and usable externally for users that want to escape all special characters in their desired match. The following special characters are escaped `. | * ? + () { } ^ $ \ : = []`

Usage

```
sanitize(x)
```

Arguments

`x` String to sanitize

Examples

```
sanitize("^")
sanitize("^+")
sanitize("^+?")
```

Index

[rx](#), [2](#)
[rx_alnum](#), [3](#)
[rx_alpha](#), [4](#)
[rx_any_of](#), [6](#)
[rx_anything](#), [4](#)
[rx_anything_but](#), [5](#)
[rx_avoid_prefix](#), [7](#)
[rx_avoid_suffix \(rx_avoid_prefix\)](#), [7](#)
[rx_begin_capture](#), [8](#)
[rx_digit](#), [8](#)
[rx_either_of](#), [9](#)
[rx_end_capture](#), [10](#)
[rx_end_of_line](#), [10](#)
[rx_find](#), [11](#)
[rx_line_break](#), [12](#)
[rx_lowercase](#), [12](#)
[rx_maybe](#), [13](#)
[rx_multiple](#), [14](#)
[rx_none_or_more](#), [14](#)
[rx_not](#), [15](#)
[rx_one_or_more](#), [16](#)
[rx_punctuation](#), [17](#)
[rx_range](#), [17](#)
[rx_seek_prefix](#), [18](#)
[rx_seek_suffix \(rx_seek_prefix\)](#), [18](#)
[rx_something](#), [19](#)
[rx_something_but](#), [19](#)
[rx_space](#), [20](#)
[rx_start_of_line](#), [21](#)
[rx_tab](#), [22](#)
[rx_uppercase](#), [22](#)
[rx_whitespace](#), [23](#)
[rx_with_any_case](#), [24](#)
[rx_word](#), [25](#)
[rx_word_char](#), [26](#)
[rx_word_edge](#), [26](#)

[sanitize](#), [27](#)