Package ‘RVerbalExpressions’

March 20, 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
NeedsCompilation no
Author Tyler Littlefield [aut, cre] (<https://orcid.org/0000-0002-6020-1125>),
Dmytro Perepolkin [ctb] (<https://orcid.org/0000-0001-8558-6183>)
Maintainer Tyler Littlefield <tylurp1@gmail.com>
Repository CRAN
Date/Publication 2024-03-20 22:20:05 UTC

R topics documented:

- rx 2
- rx_alnum 3
- rx_alpha 4
- rx_anything 4
- rx_anything_but 5
- rx_any_of 6
- rx_avoid_prefix 7
Index 28

**rx**

Constructs a Verbal Expression

**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

```r
rx()

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

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

---

**rx_alnum**

*Match alphanumeric characters.*

**Description**

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

**Usage**

```r
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 *not* match alphanumeric characters.

**Examples**

```r
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**  
*Match alphabetic characters.*

**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 *not* match alphabetic characters.

**Examples**

```r
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**  
*Match any character(s) any (including zero) number of times.*

**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**

\[rx\_anything(.data = NULL, mode = "greedy")\]
**rx_anything_but**

**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](https://www.regular-expressions.info/dot.html)

Star Quantifier: [https://www.regular-expressions.info/repeat.html](https://www.regular-expressions.info/repeat.html)

Greedy and Lazy Quantifiers: [https://www.regular-expressions.info/repeat.html#greedy](https://www.regular-expressions.info/repeat.html#greedy)

**Examples**

```r
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
```

**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**

```r
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](https://www.regular-expressions.info/charclass.html)

Examples

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

---

**rx_any_of**

Match any of these characters exactly once.

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

```r
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](https://www.regular-expressions.info/charclass.html)

Examples

```r
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_begin_capture**

*Begin a capture group.*

**Description**

Begin a capture group.

**Usage**

```r
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**

*Match a digit (0–9).*

**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**

```r
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 *not* 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

Alternatively, match either expression.

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**

End a capture group.

**Description**

End a capture group.

**Usage**

```r
grep(.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**

Match the expression only if it appears till the end of the line.

**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**

```r
grep(.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](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

Match an expression.

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
Match a line break.

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](https://codepoints.net/U+000A)
- Windows style: [https://codepoints.net/U+000D](https://codepoints.net/U+000D)
- Non printable character: [https://www.regular-expressions.info/nonprint.html](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))
```

Match lower case letters.

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 not match lower case.

**Examples**

```r
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))
```

---

**Description**

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

**Usage**

```r
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](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

*Match the previous group any number of times.*

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

*Match the previous stuff zero or many times.*

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

```r
rx_not()

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

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

Arguments

- `.data` Expression to append, typically pulled from the pipe %>%
- `value` Value to ensure absence of

References

Negative lookahead: [https://www.regular-expressions.info/lookaround.html](https://www.regular-expressions.info/lookaround.html)

Examples

```r
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

```r
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

```r
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**

*Match punctuation characters.*

### 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 not match punctuation.

### Examples

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

# create an expression
dx <- rx_punctuation()

# create input
x_string <- 'Apple 1!'

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

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

---

**rx_range**

*Match any character within the range defined by the parameters.*

### 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.
**rx_seek_prefix**

### Usage

```r
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

```r
rx_range(value = c("Var1", "Var3"))
```

### rx_seek_prefix

#### Positive lookaround functions

#### 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

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

#### Arguments

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

#### Examples

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

*Match any character(s) at least once.*

**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**

```r
rx_something(.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**

- Metacharacters: [https://www.regular-expressions.info/characters.html#special](https://www.regular-expressions.info/characters.html#special)
- Greedy and Lazy Quantifiers: [https://www.regular-expressions.info/repeat.html#greedy](https://www.regular-expressions.info/repeat.html#greedy)

**Examples**

```r
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**

*Match any character(s) except these at least once.*

**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_anythingBut()` expects *anything* including... nothing!
Usage

```r
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](https://www.regular-expressions.info/characters.html#special)

Greedy and Lazy Quantifiers: [https://www.regular-expressions.info/repeat.html#greedy](https://www.regular-expressions.info/repeat.html#greedy)

Examples

```r
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**

*Match a space character.*

---

Description

Matches a space character.

Usage

```r
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 *not* match space.
**Examples**

```r
# 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**

*Match the expression only if it appears from beginning of line.*

**Description**

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

**Usage**

```r
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**

```r
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

**Match a tab character.**

**Description**

Match a tab character.

**Usage**

```r
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 *not* match tabs.

**Details**

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

1. Tab character: [https://codepoints.net/U+0009](https://codepoints.net/U+0009)

**Examples**

```r
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

**Match upper case letters.**

**Description**

Matches upper case letters only.

**Usage**

```r
rx_uppercase(.data = NULL, inverse = FALSE)
```
**rx_whitespace**

**Arguments**

- `.data` Expression to append, typically pulled from the pipe `%%`
- `inverse` Invert match behavior, defaults to `FALSE` (match upper case). Use `TRUE` to *not* match upper case.

**Examples**

```r
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))
```

**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 *not* 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](https://codepoints.net/U+0020)
2. tab: [https://codepoints.net/U+0009](https://codepoints.net/U+0009)
3. carriage return: [https://codepoints.net/U+000D](https://codepoints.net/U+000D)
4. new line: [https://codepoints.net/U+000A](https://codepoints.net/U+000A)
5. vertical tab: [https://codepoints.net/U+000B](https://codepoints.net/U+000B)
6. form feed: [https://codepoints.net/U+000C](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 is looks for tabs with the following expression: \w+

**Usage**

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

**Arguments**

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

**Examples**

```r
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**  
*Match a word character.*

**Description**

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

**Usage**

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

**Arguments**

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

**Examples**

```r
rx_word_char()

# Same as rx_word()
```

```r
x <- rx_word_char() %>%
  rx_one_or_more()
```

---

**rx_word_edge**  
*Find beginning or end of a word.*

**Description**

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

**Usage**

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

**Arguments**

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

```r
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

```r
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