Package ‘stringi’

November 13, 2023

Version 1.8.1
Date 2023-11-09
Title Fast and Portable Character String Processing Facilities
Description A collection of character string/text/natural language processing tools for pattern searching (e.g., with 'Java'-like regular expressions or the 'Unicode' collation algorithm), random string generation, case mapping, string transliteration, concatenation, sorting, padding, wrapping, Unicode normalisation, date-time formatting and parsing, and many more. They are fast, consistent, convenient, and - thanks to 'ICU' (International Components for Unicode) - portable across all locales and platforms. Documentation about 'stringi' is provided via its website at <https://stringi.gagolewski.com/> and the paper by Gagolewski (2022, <doi:10.18637/jss.v103.i02>).

URL https://stringi.gagolewski.com/,

BugReports https://github.com/gagolews/stringi/issues

SystemRequirements ICU4C (>= 61, optional)

Type Package

Depends R (>= 3.4)
Imports tools, utils, stats

Biarch TRUE

License file LICENSE

Author Marek Gagolewski [aut, cre, cph] (<https://orcid.org/0000-0003-0637-6028>), Bartek Tartanus [ctb], and others (stringi source code); Unicode, Inc. and others (ICU4C source code, Unicode Character Database)

Maintainer Marek Gagolewski <marek@gagolewski.com>

RoxygenNote 7.2.3

Encoding UTF-8

NeedsCompilation yes

License_is_FOSS yes
Repository  CRAN
Date/Publication  2023-11-13 14:10:02 UTC

R topics documented:

- about_arguments ........................................ 4
- about_encoding .......................................... 5
- about_locale ............................................. 9
- about_search ........................................... 11
- about_search_boundaries ............................... 12
- about_search_charclass ............................... 14
- about_search_coll ..................................... 19
- about_search_fixed ................................... 20
- about_search_regex ................................... 21
- about_stringi .......................................... 25
- stri_compare ........................................... 27
- stri_count ............................................... 29
- stri_count_boundaries ................................. 31
- stri_datetime_add ...................................... 33
- stri_datetime_create .................................. 34
- stri_datetime_fields .................................. 36
- stri_datetime_format .................................. 37
- stri_datetime_fstr .................................... 41
- stri_datetime_now ..................................... 42
- stri_datetime_symbols ............................... 43
- stri_detect ............................................ 45
- stri_dup ................................................ 47
- stri_duplicated ........................................ 48
- stri_encode ............................................ 50
- stri_enc_detect ....................................... 52
- stri_enc_detect2 ...................................... 54
- stri_enc_fromutf32 .................................... 55
- stri_enc_info ......................................... 56
- stri_enc_isascii ....................................... 57
- stri_enc_isutf16be .................................... 58
- stri_enc_isutf8 ........................................ 59
- stri_enc_list .......................................... 60
- stri_enc_mark ......................................... 61
- stri_enc_set .......................................... 62
- stri_enc_toascii ...................................... 63
- stri_enc_tonative ..................................... 64
- stri_enc_toutf32 ....................................... 65
- stri_enc_toutf8 ........................................ 66
- stri_escape_unicode .................................. 67
- stri_extract_all ...................................... 68
- stri_extract_all_boundaries ....................... 72
- stri_flatten .......................................... 74
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>stri_info</td>
<td>75</td>
</tr>
<tr>
<td>stri_isempty</td>
<td>76</td>
</tr>
<tr>
<td>stri_join</td>
<td>77</td>
</tr>
<tr>
<td>stri_join_list</td>
<td>78</td>
</tr>
<tr>
<td>stri_length</td>
<td>79</td>
</tr>
<tr>
<td>stri_list2matrix</td>
<td>81</td>
</tr>
<tr>
<td>stri_locale_info</td>
<td>82</td>
</tr>
<tr>
<td>stri_locale_list</td>
<td>83</td>
</tr>
<tr>
<td>stri_locale_set</td>
<td>84</td>
</tr>
<tr>
<td>stri_locate_all</td>
<td>85</td>
</tr>
<tr>
<td>stri_locate_all_boundaries</td>
<td>89</td>
</tr>
<tr>
<td>stri_match_all</td>
<td>91</td>
</tr>
<tr>
<td>stri_na2empty</td>
<td>94</td>
</tr>
<tr>
<td>stri_numbytes</td>
<td>95</td>
</tr>
<tr>
<td>stri_opts_brkiter</td>
<td>96</td>
</tr>
<tr>
<td>stri_opts_collator</td>
<td>98</td>
</tr>
<tr>
<td>stri_opts_fixed</td>
<td>100</td>
</tr>
<tr>
<td>stri_opts_regex</td>
<td>101</td>
</tr>
<tr>
<td>stri_order</td>
<td>103</td>
</tr>
<tr>
<td>stri_pad_both</td>
<td>104</td>
</tr>
<tr>
<td>stri_rand_lipsum</td>
<td>106</td>
</tr>
<tr>
<td>stri_rand_shuffle</td>
<td>107</td>
</tr>
<tr>
<td>stri_rand_strings</td>
<td>108</td>
</tr>
<tr>
<td>stri_rank</td>
<td>109</td>
</tr>
<tr>
<td>stri_read_lines</td>
<td>111</td>
</tr>
<tr>
<td>stri_read_raw</td>
<td>112</td>
</tr>
<tr>
<td>stri_remove_empty</td>
<td>113</td>
</tr>
<tr>
<td>stri_replace_all</td>
<td>114</td>
</tr>
<tr>
<td>stri_replace_na</td>
<td>117</td>
</tr>
<tr>
<td>stri_replace_rstr</td>
<td>118</td>
</tr>
<tr>
<td>stri_reverse</td>
<td>119</td>
</tr>
<tr>
<td>stri_sort</td>
<td>120</td>
</tr>
<tr>
<td>stri_sort_key</td>
<td>121</td>
</tr>
<tr>
<td>stri_split</td>
<td>122</td>
</tr>
<tr>
<td>stri_split_boundaries</td>
<td>125</td>
</tr>
<tr>
<td>stri_split_lines</td>
<td>127</td>
</tr>
<tr>
<td>stri_sprintf</td>
<td>129</td>
</tr>
<tr>
<td>stri_startswith</td>
<td>131</td>
</tr>
<tr>
<td>stri_stats_general</td>
<td>134</td>
</tr>
<tr>
<td>stri_stats_latex</td>
<td>135</td>
</tr>
<tr>
<td>stri_sub</td>
<td>136</td>
</tr>
<tr>
<td>stri_subset</td>
<td>139</td>
</tr>
<tr>
<td>stri_sub_all</td>
<td>141</td>
</tr>
<tr>
<td>stri_timezone_get</td>
<td>143</td>
</tr>
<tr>
<td>stri_timezone_info</td>
<td>144</td>
</tr>
<tr>
<td>stri_timezone_list</td>
<td>146</td>
</tr>
<tr>
<td>stri_trans_char</td>
<td>147</td>
</tr>
<tr>
<td>stri_trans_general</td>
<td>148</td>
</tr>
</tbody>
</table>
about_arguments

Description

Below we explain how `stringi` deals with its functions’ arguments.

If some function violates one of the following rules (for a very important reason), this is clearly indicated in its documentation (with discussion).

Coercion of Arguments

When a character vector argument is expected, factors and other vectors coercible to characters vectors are silently converted with `as.character`, otherwise an error is generated. Coercion from a list which does not consist of length-1 atomic vectors issues a warning.

When a logical, numeric, or integer vector argument is expected, factors are converted with `as.*((as.character(...))`, and other coercible vectors are converted with `as.*`, otherwise an error is generated.

Vectorization

Almost all functions are vectorized with respect to all their arguments and the recycling rule is applied whenever necessary. Due to this property you may, for instance, search for one pattern in each given string, search for each pattern in one given string, and search for the i-th pattern within the i-th string.

We of course took great care of performance issues: e.g., in regular expression searching, regex matchers are reused from iteration to iteration, as long as it is possible.

Functions with some non-vectorized arguments are rare: e.g., regular expression matcher’s settings are established once per each call.

Some functions assume that a vector with one element is given as an argument (like `collapse` in `stri_join`). In such cases, if an empty vector is given you will get an error and for vectors with more than 1 elements - a warning will be generated (only the first element will be used).

You may find details on vectorization behavior in the man pages on each particular function of your interest.
Handling Missing Values (NAs)

`stringi` handles missing values consistently. For any vectorized operation, if at least one vector element is missing, then the corresponding resulting value is also set to `NA`.

Preserving Object Attributes

Generally, all our functions drop input objects' attributes (e.g., names, dim, etc.). This is due to deep vectorization as well as for efficiency reasons. If the preservation of attributes is needed, important attributes can be manually copied. Alternatively, the notation `x[] <- stri_...(x, ...)` can sometimes be used too.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other stringi_general_topics: about_encoding, about_locale, about_search_boundaries, about_search_charclass, about_search_coll, about_search_fixed, about_search_regex, about_search, about_stringi

Description

This manual page explains how `stringi` deals with character strings in various encodings.

In particular we should note that:

- R lets strings in ASCII, UTF-8, and your platform’s native encoding coexist. A character vector printed on the console by calling `print` or `cat` is silently re-encoded to the native encoding.
- Functions in `stringi` process each string internally in Unicode, the most universal character encoding ever. Even if a string is given in the native encoding, i.e., your platform’s default one, it will be converted to Unicode (precisely: UTF-8 or UTF-16).
- Most `stringi` functions always return UTF-8 encoded strings, regardless of the input encoding. What is more, the functions have been optimized for UTF-8/ASCII input (they have competitive, if not better performance, especially when performing more complex operations like string comparison, sorting, and even concatenation). Thus, it is best to rely on cascading calls to `stringi` operations solely.
Details

Quoting the ICU User Guide, 'Hundreds of encodings have been developed over the years, each for small groups of languages and for special purposes. As a result, the interpretation of text, input, sorting, display, and storage depends on the knowledge of all the different types of character sets and their encodings. Programs have been written to handle either one single encoding at a time and switch between them, or to convert between external and internal encodings.'

'Unicode provides a single character set that covers the major languages of the world, and a small number of machine-friendly encoding forms and schemes to fit the needs of existing applications and protocols. It is designed for best interoperability with both ASCII and ISO-8859-1 (the most widely used character sets) to make it easier for Unicode to be used in almost all applications and protocols' (see the ICU User Guide).

The Unicode Standard determines the way to map any possible character to a numeric value – a so-called code point. Such code points, however, have to be stored somehow in computer’s memory. The Unicode Standard encodes characters in the range U+0000..U+10FFFF, which amounts to a 21-bit code space. Depending on the encoding form (UTF-8, UTF-16, or UTF-32), each character will then be represented either as a sequence of one to four 8-bit bytes, one or two 16-bit code units, or a single 32-bit integer (compare the ICU FAQ).

Unicode can be thought of as a superset of the spectrum of characters supported by any given code page.

UTF-8 and UTF-16

For portability reasons, the UTF-8 encoding is the most natural choice for representing Unicode character strings in R. UTF-8 has ASCII as its subset (code points 1–127 represent the same characters in both of them). Code points larger than 127 are represented by multi-byte sequences (from 2 to 4 bytes: Please note that not all sequences of bytes are valid UTF-8, compare stri_enc_isutf8).

Most of the computations in stringi are performed internally using either UTF-8 or UTF-16 encodings (this depends on type of service you request: some ICU services are designed only to work with UTF-16). Due to such a choice, with stringi you get the same result on each platform, which is – unfortunately – not the case of base R’s functions (for instance, it is known that performing a regular expression search under Linux on some texts may give you a different result to those obtained under Windows). We really had portability in our minds while developing our package!

We have observed that R correctly handles UTF-8 strings regardless of your platform’s native encoding (see below). Therefore, we decided that most functions in stringi will output its results in UTF-8 – this speeds up computations on cascading calls to our functions: the strings does not have to be re-encoded each time.

Note that some Unicode characters may have an ambiguous representation. For example, “a with ogonek” (one character) and “a”+“ogonek” (two graphemes) are semantically the same. stringi provides functions to normalize character sequences, see stri_trans_nfc for discussion. However, it is observed that denormalized strings do appear very rarely in typical string processing activities.

Additionally, do note that stringi silently removes byte order marks (BOMs - they may incidentally appear in a string read from a text file) from UTF8-encoded strings, see stri_enc_toutf8.

Character Encodings in R

Data in memory are just bytes (small integer values) – an encoding is a way to represent characters with such numbers, it is a semantic ‘key’ to understand a given byte sequence. For example, in ISO-
8859-2 (Central European), the value 177 represents Polish “a with ogonek”, and in ISO-8859-1 (Western European), the same value denotes the “plus-minus” sign. Thus, a character encoding is a translation scheme: we need to communicate with R somehow, relying on how it represents strings.

Overall, R has a very simple encoding marking mechanism, see `stri_enc_mark`. There is an implicit assumption that your platform’s default (native) encoding always extends ASCII – `stringi` checks that whenever your native encoding is being detected automatically on ICU’s initialization and each time when you change it manually by calling `stri_enc_set`.

Character strings in R (internally) can be declared to be in:

- UTF-8;
- latin1, i.e., either ISO-8859-1 (Western European on Linux, OS X, and other Unixes) or WINDOWS-1252 (Windows);
- bytes – for strings that should be manipulated as sequences of bytes.

Moreover, there are two other cases:

- ASCII – for strings consisting only of byte codes not greater than 127;
- native (a.k.a. unknown in Encoding; quite a misleading name: no explicit encoding mark) – for strings that are assumed to be in your platform’s native (default) encoding. This can represent UTF-8 if you are an OS X user, or some 8-bit Windows code page, for example. The native encoding used by R may be determined by examining the LC_CTYPE category, see `Sys.getlocale`.

Intuitively, “native” strings result from reading a string from stdin (e.g., keyboard input). This makes sense: your operating system works in some encoding and provides R with some data.

Each time when a `stringi` function encounters a string declared in native encoding, it assumes that the input data should be translated from the default encoding, i.e., the one returned by `stri_enc_get` (unless you know what you are doing, the default encoding should only be changed if the automatic encoding detection process fails on `stringi` load).

Functions which allow ‘bytes’ encoding markings are very rare in `stringi`, and were carefully selected. These are: `stri_enc_toutf8` (with argument `is_unknown_8bit=TRUE`), `stri_enc_toascii`, and `stri_encode`.

Finally, note that R lets strings in ASCII, UTF-8, and your platform’s native encoding coexist. A character vector printed with `print`, `cat`, etc., is silently re-encoded so that it can be properly shown, e.g., on the console.

### Encoding Conversion

Apart from automatic conversion from the native encoding, you may re-encode a string manually, for example when you read it from a file created on a different platform. Call `stri_enc_list` for the list of encodings supported by ICU. Note that converter names are case-insensitive and ICU tries to normalize the encoding specifiers. Leading zeroes are ignored in sequences of digits (if further digits follow), and all non-alphanumeric characters are ignored. Thus the strings ‘UTF-8’, ‘utf_8’, ‘u*Tf08’ and ‘Utf 8’ are equivalent.

The `stri_encode` function allows you to convert between any given encodings (in some cases you will obtain bytes-marked strings, or even lists of raw vectors (i.e., for UTF-16). There are also some useful more specialized functions, like `stri_enc_toutf32` (converts a character vector
to a list of integers, where one code point is exactly one numeric value) or \texttt{stri_enc_toascii} (substitutes all non-ASCII bytes with the SUBSTITUTE CHARACTER, which plays a similar role as \texttt{R}'s \texttt{NA} value).

There are also some routines for automated encoding detection, see, e.g., \texttt{stri_enc_detect}.

**Encoding Detection**

Given a text file, one has to know how to interpret (encode) raw data in order to obtain meaningful information.

Encoding detection is always an imprecise operation and needs a considerable amount of data. However, in case of some encodings (like UTF-8, ASCII, or UTF-32) a “false positive” byte sequence is quite rare (statistically speaking).

Check out \texttt{stri_enc_detect} (among others) for a useful function in this category.

**Author(s)**

Marek Gagolewski and other contributors

**References**


**See Also**

The official online manual of \texttt{stringi} at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., \texttt{stringi}: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other \texttt{stringi_general_topics}: \texttt{about_arguments}, \texttt{about_locale}, \texttt{about_search_boundaries}, \texttt{about_search_charclass}, \texttt{about_search_coll}, \texttt{about_search_fixed}, \texttt{about_search_regex}, \texttt{about_search}, \texttt{about_stringi}

Other \texttt{encoding_management}: \texttt{stri_enc_info()}, \texttt{stri_enc_list()}, \texttt{stri_enc_mark()}, \texttt{stri_enc_set()}

Other \texttt{encoding_detection}: \texttt{stri_enc_detect2()}, \texttt{stri_enc_detect()}, \texttt{stri_enc_isascii()}, \texttt{stri_enc_isutf16be()}, \texttt{stri_enc_isutf8()}

Other \texttt{encoding_conversion}: \texttt{stri_enc_fromutf32()}, \texttt{stri_enc_toascii()}, \texttt{stri_enc_tonative()}, \texttt{stri_enc_toutf32()}, \texttt{stri_enc_toutf8()}, \texttt{stri_encode()}


**Description**

In this section we explain how we specify locales in `stringi`. Locale is a fundamental concept in ICU. It identifies a specific user community, i.e., a group of users who have similar culture and language expectations for human-computer interaction.

**Details**

Because a locale is just an identifier of a region, no validity check is performed when you specify a Locale. ICU is implemented as a set of services. If you want to verify whether particular resources are available in the locale you asked for, you must query those resources. Note: when you ask for a resource for a particular locale, you get back the best available match, not necessarily precisely the one you requested.

**Locale Identifiers**

ICU services are parametrized by locale, to deliver culturally correct results. Locales are identified by character strings of the form `Language code, Language_Country code, or Language_Country_Variant code`, e.g., 'en_US'.

The two-letter `Language code` uses the ISO-639-1 standard, e.g., 'en' stands for English, 'pl' – Polish, 'fr' – French, and 'de' for German.

Country is a two-letter code following the ISO-3166 standard. This is to reflect different language conventions within the same language, for example in US-English ('en_US') and Australian-English ('en_AU').

Differences may also appear in language conventions used within the same country. For example, the Euro currency may be used in several European countries while the individual country’s currency is still in circulation. In such a case, ICU Variant '_EURO' could be used for selecting locales that support the Euro currency.

The final (optional) element of a locale is a list of keywords together with their values. Keywords must be unique. Their order is not significant. Unknown keywords are ignored. The handling of keywords depends on the specific services that utilize them. Currently, the following keywords are recognized: `calendar, collation, currency, and numbers`, e.g., `fr@collation=phonebook;calendar=islamic-civil` is a valid French locale specifier together with keyword arguments. For more information, refer to the ICU user guide.

For a list of locales that are recognized by ICU, call `stri_locale_list`.

Note that in `stringi`, ‘C’ is a synonym of ‘en_US_POSIX’.

**A Note on Default Locales**

Each locale-sensitive function in `stringi` selects the current default locale if an empty string or `NULL` is provided as its `locale` argument. Default locales are available to all the functions; initially, the system locale on that platform is used, but it may be changed by calling `stri_locale_set`. 
Your program should avoid changing the default locale. All locale-sensitive functions may request any desired locale per-call (by specifying the `locale` argument), i.e., without referencing to the default locale. During many tests, however, we did not observe any improper behavior of `stringi` while using a modified default locale.

**Locale-Sensitive Functions in stringi**

One of many examples of locale-dependent services is the Collator, which performs a locale-aware string comparison. It is used for string comparing, ordering, sorting, and searching. See `stri_opts_collator` for the description on how to tune its settings, and its `locale` argument in particular.

When choosing a resource bundle that is not available in the explicitly requested locale (but not when using the default locale) nor in its more general variants (e.g., `es_ES` vs `es`), a warning is emitted.

Other locale-sensitive functions include, e.g., `stri_trans_tolower` (that does character case mapping).

**Author(s)**

Marek Gagolewski and other contributors

**References**


**See Also**

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other locale_management: `stri_locale_info()`, `stri_locale_list()`, `stri_locale_set()`

Other locale_sensitive: `%s<%()`, `about_search_boundaries`, `about_search_coll`, `stri_compare()`, `stri_count_boundaries()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_collator()`, `stri_order()`, `stri_rank()`, `stri_sort_key()`, `stri_sort()`, `stri_split_boundaries()`, `stri_trans_tolower()`, `stri_unique()`, `stri_wrap()`

Other stringi_general_topics: `about_arguments`, `about_encoding`, `about_search_boundaries`, `about_search_charclass`, `about_search_coll`, `about_search_fixed`, `about_search_regex`, `about_search`, `about_stringi`
Description

This man page explains how to perform string search-based operations in *stringi*.

Details

The following independent string searching engines are available in *stringi*:

- `stri_*_regex` – ICU’s regular expressions (regexes), see about_search_regex,
- `stri_*_fixed` – locale-independent byte-wise pattern matching, see about_search_fixed,
- `stri_*_coll` – ICU’s *StringSearch*, locale-sensitive, Collator-based pattern search, useful for natural language processing tasks, see about_search_coll,
- `stri_*_charclass` – character classes search, e.g., Unicode General Categories or Binary Properties, see about_search_charclass,
- `stri_*_boundaries` – text boundary analysis, see about_search_boundaries

Each search engine is able to perform many search-based operations. These may include:

- `stri_detect_*` - detect if a pattern occurs in a string, see, e.g., *stri_detect*.
- `stri_count_*` - count the number of pattern occurrences, see, e.g., *stri_count*.
- `stri_locate_*` - locate all, first, or last occurrences of a pattern, see, e.g., *stri_locate*.
- `stri_extract_*` - extract all, first, or last occurrences of a pattern, see, e.g., *stri_extract* and, in case of regexes, *stri_match*.
- `stri_replace_*` - replace all, first, or last occurrences of a pattern, see, e.g., *stri_replace* and also *stri_trim*.
- `stri_split_*` - split a string into chunks indicated by occurrences of a pattern, see, e.g., *stri_split*.
- `stri_startswith_*` and `stri_endswith_*` detect if a string starts or ends with a pattern match, see, e.g., *stri_startswith*.
- `stri_subset_*` - return a subset of a character vector with strings that match a given pattern, see, e.g., *stri_subset*.

Author(s)

Marek Gagolewski and other contributors
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other text_boundaries: about_search_boundaries, stri_count_boundaries(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_brkiter, stri_split_boundaries(), stri_split_lines(), stri_trans_tolower(), stri_wrap()

Other search_regex: about_search_regex, stri_opts_regex()

Other search_fixed: about_search_fixed, stri_opts_fixed()

Other search_coll: about_search_coll, stri_opts_collator()

Other search_charclass: about_search_charclass, stri_trim_both()

Other search_detect: stri_detect(), stri_startswith()

Other search_count: stri_count_boundaries(), stri_count()

Other search_locate: stri_locate_all_boundaries(), stri_locate_all()

Other search_replace: stri_replace_all(), stri_replace_rstr(), stri_trim_both()

Other search_split: stri_split_boundaries(), stri_split_lines(), stri_split()

Other search_subset: stri_subset()

Other search_extract: stri_extract_all_boundaries(), stri_extract_all(), stri_match_all()

Other stringi_general_topics: about_arguments, about_encoding, about_locale, about_search_boundaries, about_search_charclass, about_search_coll, about_search_fixed, about_search_regex, about_stringi
• Located a particular unit of the text (for example, finding the third word in the document), see `stri_locate_all_boundaries`.

Generally, text boundary analysis is a locale-dependent operation. For example, in Japanese and Chinese one does not separate words with spaces - a line break can occur even in the middle of a word. These languages have punctuation and diacritical marks that cannot start or end a line, so this must also be taken into account.

`stringi` uses ICU’s `BreakIterator` to locate specific text boundaries. Note that the `BreakIterator`’s behavior may be controlled in some cases, see `stri_opts_brkiter`.

• The character boundary iterator tries to match what a user would think of as a “character” – a basic unit of a writing system for a language – which may be more than just a single Unicode code point.
• The word boundary iterator locates the boundaries of words, for purposes such as “Find whole words” operations.
• The line_break iterator locates positions that would be appropriate to wrap lines when displaying the text.
• The break iterator of type sentence locates sentence boundaries.

For technical details on different classes of text boundaries refer to the ICU User Guide, see below.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other `locale_sensitive`: `%s<%`, `about_locale`, `about_search_coll`, `stri_compare()`, `stri_count_boundaries()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_collator()`, `stri_order()`, `stri_rank()`, `stri_sort_key()`, `stri_sort()`, `stri_split_boundaries()`, `stri_trans_tolower()`, `stri_unique()`, `stri_wrap()`

Other `text_boundaries`: `about_search`, `stri_count_boundaries()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_brkiter()`, `stri_split_boundaries()`, `stri_split_lines()`, `stri_trans_tolower()`, `stri_wrap()`

Other `stringi_general_topics`: `about_arguments`, `about_encoding`, `about_locale`, `about_search_charclass`, `about_search_coll`, `about_search_fixed`, `about_search_regex`, `about_search`, `about_stringi`
about_search_charclass

Character Classes in stringi

Description

Here we describe how character classes (sets) can be specified in the stringi package. These are useful for defining search patterns (note that the ICU regex engine uses the same scheme for denoting character classes) or, e.g., generating random code points with stri_rand_strings.

Details

All stri_*_charclass functions in stringi perform a single character (i.e., Unicode code point) search-based operations. You may obtain the same results using about_search_regex. However, these very functions aim to be faster.

Character classes are defined using ICU’s UnicodeSet patterns. Below we briefly summarize their syntax. For more details refer to the bibliographic References below.

UnicodeSet patterns

A UnicodeSet represents a subset of Unicode code points (recall that stringi converts strings in your native encoding to Unicode automatically). Legal code points are U+0000 to U+10FFFF, inclusive.

Patterns either consist of series of characters bounded by square brackets (such patterns follow a syntax similar to that employed by regular expression character classes) or of Perl-like Unicode property set specifiers.

[] denotes an empty set, [a] – a set consisting of character “a”, \[u0105\] – a set with character U+0105, and [abc] – a set with “a”, “b”, and “c”.

[a-z] denotes a set consisting of characters “a” through “z” inclusively, in Unicode code point order.

Some set-theoretic operations are available. ^ denotes the complement, e.g., [^a-z] contains all characters but “a” through “z”. Moreover, [[pat1][pat2]], [[pat1]\&[pat2]], and [[pat1]-[pat2]] denote union, intersection, and asymmetric difference of sets specified by pat1 and pat2, respectively.

Note that all white-spaces are ignored unless they are quoted or back-slashed (white spaces can be freely used for clarity, as [a c d-f m] means the same as [acd-fm]). stringi does not allow including multi-character strings (see UnicodeSet API documentation). Also, empty string patterns are disallowed.

Any character may be preceded by a backslash in order to remove its special meaning.

A malformed pattern always results in an error.

Set expressions at a glance (according to https://unicode-org.github.io/icu/userguide/strings/regexp.html):

Some examples:
about_search_charclass

\[abc\] Match any of the characters a, b or c.
\[^{abc}\] Negation – match any character except a, b or c.
\[A-M\] Range – match any character from A to M. The characters to include are determined by Unicode code point ordering.
\[\u0000-\u0010ffff\] Range – match all characters.
\[\p{Letter}\] or \[\p{General_Category=Letter}\] or \[\p{L}\] Characters with Unicode Category = Letter. All forms shown are equivalent.
\[\P{Letter}\] Negated property (Note the upper case \P) – match everything except Letters.
\[\p{numeric_value=9}\] Match all numbers with a numeric value of 9. Any Unicode Property may be used in set expressions.
\[\p{Letter}\&\p{script=cyrillic}\] Set intersection – match the set of all Cyrillic letters.
\[\p{Letter}\-\p{script=latin}\] Set difference – match all non-Latin letters.
\[\[a-z][A-Z][0-9]\] or \[a-zA-Z0-9\] Implicit union of sets – match ASCII letters and digits (the two forms are equivalent).
\[:^{script=Greek}:\] Alternative POSIX-like syntax for properties – equivalent to \p{script=Greek}.

Unicode properties

Unicode property sets are specified with a POSIX-like syntax, e.g., \[:Letter:], or with a (extended) Perl-style syntax, e.g., \p{L}. The complements of the above sets are \[^{Letter}:\] and \P{L}, respectively.

The names are normalized before matching (for example, the match is case-insensitive). Moreover, many names have short aliases.

Among predefined Unicode properties we find, e.g.:

- Unicode General Categories, e.g., Lu for uppercase letters,
- Unicode Binary Properties, e.g., WHITE_SPACE,

and many more (including Unicode scripts).

Each property provides access to the large and comprehensive Unicode Character Database. Generally, the list of properties available in ICU is not well-documented. Please refer to the References section for some links.

Please note that some classes might overlap. However, e.g., General Category Z (some space) and Binary Property WHITE_SPACE matches different character sets.

Unicode General Categories

The Unicode General Category property of a code point provides the most general classification of that code point. Each code point falls into one and only one Category.

Cc  a C0 or C1 control code.
Cf  a format control character.
Cn  a reserved unassigned code point or a non-character.
Co  a private-use character.
Cs  a surrogate code point.
Lc  the union of Lu, Ll, Lt.
Ll  a lowercase letter.
Lm  a modifier letter.
Lo  other letters, including syllables and ideographs.
Lt  a digraphic character, with the first part uppercase.
Lu  an uppercase letter.
Mc  a spacing combining mark (positive advance width).
Me  an enclosing combining mark.
Mn  a non-spacing combining mark (zero advance width).
Nd  a decimal digit.
Nl  a letter-like numeric character.
No  a numeric character of other type.
Pd  a dash or hyphen punctuation mark.
Ps  an opening punctuation mark (of a pair).
Pc  a connecting punctuation mark, like a tie.
Po  a punctuation mark of other type.
Pi  an initial quotation mark.
Pf  a final quotation mark.
Sm  a symbol of mathematical use.
Sc  a currency sign.
Sk  a non-letter-like modifier symbol.
So  a symbol of other type.
Zs  a space character (of non-zero width).
Zl  U+2028 LINE SEPARATOR only.
Zp  U+2029 PARAGRAPH SEPARATOR only.
C  the union of Cc, Cf, Cs, Co, Cn.
L  the union of Lu, Ll, Lt, Lm, Lo.
M  the union of Mn, Mc, Me.
N  the union of Nd, Nl, No.
P  the union of Pc, Pd, Ps, Pe, Pi, Pf, Po.
S  the union of Sm, Sc, Sk, So.
Z  the union of Zs, Zl, Zp
Unicode Binary Properties

Each character may follow many Binary Properties at a time. Here is a comprehensive list of supported Binary Properties:

- **ALPHABETIC**  alphabetic character.
- **ASCII_HEX_DIGIT**  a character matching the \([0-9A-Fa-f]\) charclass.
- **BIDI_CONTROL**  a format control which have specific functions in the Bidi (bidirectional text) Algorithm.
- **BIDI_MIRRORED**  a character that may change display in right-to-left text.
- **DASH**  a kind of a dash character.
- **DEFAULT_IGNORABLE_CODE_POINT**  characters that are ignorable in most text processing activities, e.g., <2060..206F, FFF0..FFFF, E0000..E0FFF>.
- **DEPRECATED**  a deprecated character according to the current Unicode standard (the usage of deprecated characters is strongly discouraged).
- **DIACRITIC**  a character that linguistically modifies the meaning of another character to which it applies.
- **EXTENDER**  a character that extends the value or shape of a preceding alphabetic character, e.g., a length and iteration mark.
- **HEX_DIGIT**  a character commonly used for hexadecimal numbers, see also ASCII_HEX_DIGIT.
- **HYPHEN**  a dash used to mark connections between pieces of words, plus the Katakana middle dot.
- **ID_CONTINUE**  a character that can continue an identifier, ID_START+Mn+Mc+Nd+Pc.
- **ID_START**  a character that can start an identifier, Lu+Ll+Lt+Lm+Lo+Nl.
- **IDEOGRAPHIC**  a CJKV (Chinese-Japanese-Korean-Vietnamese) ideograph.
- **LOWERCASE** ...
- **MATH** ...
- **NONCHARACTER_CODE_POINT** ...
- **QUOTATION_MARK** ...
- **SOFT_DOTTED**  a character with a “soft dot”, like i or j, such that an accent placed on this character causes the dot to disappear.
- **TERMINAL_PUNCTUATION**  a punctuation character that generally marks the end of textual units.
- **UPPERCASE** ...
- **WHITE_SPACE**  a space character or TAB or CR or LF or ZWSP or ZWNBS.
CASE_IGNORABLE ...
CHANGES_WHEN_LOWERCASED ...
CHANGES_WHEN_UPPERCASED ...
CHANGES_WHEN_TITLECASED ...
CHANGES_WHEN_CASEFOLDED ...
CHANGES_WHEN_CASEMAPPED ...
CHANGES_WHEN_NFKC_CASEFOLDED ...
EMOJI Since ICU 57
EMOJI_PRESENTATION Since ICU 57
EMOJI_MODIFIER Since ICU 57
EMOJI_MODIFIER_BASE Since ICU 57

POSIX Character Classes

Avoid using POSIX character classes, e.g., [:punct:]. The ICU User Guide (see below) states that in general they are not well-defined, so you may end up with something different than you expect.

In particular, in POSIX-like regex engines, [:punct:] stands for the character class corresponding to the ispunct() classification function (check out man 3 ispunct on UNIX-like systems). According to ISO/IEC 9899:1990 (ISO C90), the ispunct() function tests for any printing character except for space or a character for which isalnum() is true. However, in a POSIX setting, the details of what characters belong into which class depend on the current locale. So the [:punct:] class does not lead to a portable code (again, in POSIX-like regex engines).

Therefore, a POSIX flavor of [:punct:] is more like \[\p{P}\p{S}] in ICU. You have been warned.

Author(s)

Marek Gagolewski and other contributors

References

*The Unicode Character Database – Unicode Standard Annex #44, [https://www.unicode.org/reports/tr44/](https://www.unicode.org/reports/tr44/)*
*Unicode Script Data, [https://www.unicode.org/Public/UNIDATA/Scripts.txt](https://www.unicode.org/Public/UNIDATA/Scripts.txt)*
*icu::Unicodeset Class Reference – ICU4C API Documentation, [https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1UnicodeSet.html](https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1UnicodeSet.html)*
Locale-Sensitive Text Searching in stringi

Description

String searching facilities described here provide a way to locate a specific piece of text. Interestingly, locale-sensitive searching, especially on a non-English text, is a much more complex process than it seems at first glance.

Locale-Aware String Search Engine

All `stri_*_coll` functions in `stringi` use ICU’s StringSearch engine, which implements a locale-sensitive string search algorithm. The matches are defined by using the notion of “canonical equivalence” between strings.

Tuning the Collator’s parameters allows you to perform correct matching that properly takes into account accented letters, conjoined letters, ignorable punctuation and letter case.

For more information on ICU’s Collator and the search engine and how to tune it up in `stringi`, refer to `stri_opts_collator`.

Please note that ICU’s StringSearch-based functions are often much slower that those to perform fixed pattern searches.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_coll: about_search, stri_opts_collator()

Other locale_sensitive: %s<%(), about_locale, about_search_boundaries, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_collator(), stri_order(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()

Other stringi_general_topics: about_arguments, about_encoding, about_locale, about_search_boundaries, about_search_charclass, about_search_fixed, about_search_regex, about_search, about_stringi

about_search_fixed Locale-Insensitive Fixed Pattern Matching in stringi

Description

String searching facilities described here provide a way to locate a specific sequence of bytes in a string. The search engine’s settings may be tuned up (for example to perform case-insensitive search) via a call to the stri_opts_fixed function.

Byte Compare

The fast Knuth-Morris-Pratt search algorithm, with worst time complexity of O(n+p) (n == length(str), p == length(pattern)) is implemented (with some tweaks for very short search patterns).

Be aware that, for natural language processing, fixed pattern searching might not be what you actually require. It is because a bitwise match will not give correct results in cases of:

1. accented letters;
2. conjoined letters;
3. ignorable punctuation;
4. ignorable case,

see also about_search_coll.

Note that the conversion of input data to Unicode is done as usual.

Author(s)

Marek Gagolewski and other contributors
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_fixed: about_search, stri_opts_fixed()

Other stringi_general_topics: about_arguments, about_encoding, about_locale, about_search_boundaries, about_search_charclass, about_search_coll, about_search_regex, about_search, about_stringi

---

**about_search_regex**  
*Regular Expressions in stringi*

**Description**

A regular expression is a pattern describing, possibly in a very abstract way, a text fragment. With so many regex functions in stringi, regular expressions may be a very powerful tool to perform string searching, substring extraction, string splitting, etc., tasks.

**Details**

All stri_*_regex functions in stringi use the ICU regex engine. Its settings may be tuned up (for example to perform case-insensitive search) via the stri_opts_regex function.

Regular expression patterns in ICU are quite similar in form and behavior to Perl’s regexes. Their implementation is loosely inspired by JDK 1.4 java.util.regex. ICU Regular Expressions conform to the Unicode Technical Standard #18 (see References section) and its features are summarized in the ICU User Guide (see below). A good general introduction to regexes is (Friedl, 2002). Some general topics are also covered in the R manual, see regex.

**ICU Regex Operators at a Glance**

Here is a list of operators provided by the ICU User Guide on regexes.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>|</td>
<td>Alternation. A</td>
</tr>
<tr>
<td>*</td>
<td>Match 0 or more times. Match as many times as possible.</td>
</tr>
<tr>
<td>+</td>
<td>Match 1 or more times. Match as many times as possible.</td>
</tr>
<tr>
<td>?</td>
<td>Match zero or one times. Prefer one.</td>
</tr>
<tr>
<td>{n}</td>
<td>Match exactly n times.</td>
</tr>
<tr>
<td>{n,}</td>
<td>Match at least n times. Match as many times as possible.</td>
</tr>
<tr>
<td>{n,m}</td>
<td>Match between n and m times. Match as many times as possible, but not more than m.</td>
</tr>
<tr>
<td>*?</td>
<td>Match 0 or more times. Match as few times as possible.</td>
</tr>
<tr>
<td>+?</td>
<td>Match 1 or more times. Match as few times as possible.</td>
</tr>
<tr>
<td>??</td>
<td>Match zero or one times. Prefer zero.</td>
</tr>
<tr>
<td>{n}?</td>
<td>Match exactly n times.</td>
</tr>
</tbody>
</table>
{n,}?  Match at least n times, but no more than required for an overall pattern match.
{n,m}?  Match between n and m times. Match as few times as possible, but not less than n.
++  Match 0 or more times. Match as many times as possible when first encountered, do not retry
    with fewer even if overall match fails (Possessive Match).
++  Match 1 or more times. Possessive match.
?+  Match zero or one times. Possessive match.
{n}+  Match exactly n times.
{n,}+  Match at least n times. Possessive Match.
{n,m}+  Match between n and m times. Possessive Match.
(...)  Capturing parentheses. Range of input that matched the parenthesized sub-expression is
        available after the match, see stri_match.
(?:...)  Non-capturing parentheses. Groups the included pattern, but does not provide capturing
        of matching text. Somewhat more efficient than capturing parentheses.
(?<...)  Atomic-match parentheses. The first match of the parenthesized sub-expression is the
        only one tried; if it does not lead to an overall pattern match, back up the search for a match
to a position before the (?).  
(?#...)  Free-format comment (?# comment).
(?=...)  Look-ahead assertion. True if the parenthesized pattern matches at the current input po-
        sition, but does not advance the input position.
(?!...)  Negative look-ahead assertion. True if the parenthesized pattern does not match at the
        current input position. Does not advance the input position.
(?<=...)  Look-behind assertion. True if the parenthesized pattern matches text preceding the
        current input position, with the last character of the match being the input character just before
        the current position. Does not alter the input position. The length of possible strings matched
        by the look-behind pattern must not be unbounded (no * or + operators.)
(?<!...)  Negative Look-behind assertion. True if the parenthesized pattern does not match text
        preceding the current input position, with the last character of the match being the input char-
        acter just before the current position. Does not alter the input position. The length of possible
        strings matched by the look-behind pattern must not be unbounded (no * or + operators.)
(?<name>...)  Named capture group, where name (enclosed within the angle brackets) is a se-
        quence like [A-Za-z][A-Za-z0-9]*
(?ismwx-ismwx:...)  Flag settings. Evaluate the parenthesized expression with the specified flags
        enabled or -disabled, see also stri_opts_regex.
(?ismwxt-ismwtx)  Flag settings. Change the flag settings. Changes apply to the portion of the
        pattern following the setting. For example, (?i) changes to a case insensitive match, see also
        stri_opts_regex.

ICU Regex Meta-characters at a Glance

Here is a list of meta-characters provided by the ICU User Guide on regexes.

\a  Match a BELL, \u0007.

\A  Match at the beginning of the input. Differs from *, in that \A will not match after a new line
    within the input.
\b Match if the current position is a word boundary. Boundaries occur at the transitions between word (\w) and non-word (\W) characters, with combining marks ignored. For better word boundaries, see ICU Boundary Analysis, e.g., \texttt{stri_extract_all_words}.

\B Match if the current position is not a word boundary.

\cX Match a control-\textbackslash X character.

\d Match any character with the Unicode General Category of Nd (Number, Decimal Digit.).

\D Match any character that is not a decimal digit.

\e Match an ESCAPE, \textbackslash u001B.

\E Terminates a \texttt{\Q ... \E} quoted sequence.

\f Match a FORM FEED, \textbackslash u000C.

\g Match if the current position is at the end of the previous match.

\h Match a Horizontal White Space character. They are characters with Unicode General Category of Space_Separator plus the ASCII tab, \textbackslash u0009. [Since ICU 55]

\H Match a non-Horizontal White Space character. [Since ICU 55]

\k<name> Named Capture Back Reference. [Since ICU 55]

\l Match a LINE FEED, \textbackslash u000A.

\l\{UNICODE CHARACTER NAME\} Match the named character.

\l\{UNICODE PROPERTY NAME\} Match any character with the specified Unicode Property.

\L\{UNICODE PROPERTY NAME\} Match any character not having the specified Unicode Property.

\Q Quotes all following characters until \E.

\r Match a CARRIAGE RETURN, \textbackslash u000D.

\s Match a white space character. White space is defined as [\t\n\f\r\p{Z}].

\S Match a non-white space character.

\t Match a HORIZONTAL TABULATION, \textbackslash u0009.

\uhhhhh Match the character with the hex value hhhh.

\Uhhhhhhhh Match the character with the hex value hhhhhhhh. Exactly eight hex digits must be provided, even though the largest Unicode code point is \textbackslash u0010ffff.

\w Match a word character. Word characters are [\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctuation}\u200c\u200d].

\W Match a non-word character.

\x{hhhh} Match the character with hex value hhhh. From one to six hex digits may be supplied.

\xhh Match the character with two digit hex value hh

\X Match a Grapheme Cluster.

\Z Match if the current position is at the end of input, but before the final line terminator, if one exists.

\z Match if the current position is at the end of input.

\n Back Reference. Match whatever the nth capturing group matched. n must be a number > 1 and < total number of capture groups in the pattern.

\0ooo Match an Octal character. 'ooo' is from one to three octal digits. 0377 is the largest allowed Octal character. The leading zero is required; it distinguishes Octal constants from back references.
[pattern] Match any one character from the set.

. Match any character except for - by default - newline, compare stri_opts_regex.
^ Match at the beginning of a line.
$ Match at the end of a line.
\ [outside of sets] Quotes the following character. Characters that must be quoted to be treated as literals are * ? + [ ( ) { } ^ $ | \ .
\ [inside sets] Quotes the following character. Characters that must be quoted to be treated as literals are [ ] \; Characters that may need to be quoted, depending on the context are ~ &.

Character Classes

The syntax is similar, but not 100% compatible with the one described in about_search_charclass. In particular, whitespaces are not ignored and set-theoretic operations are denoted slightly differently. However, other than this about_search_charclass is a good reference on the capabilities offered.
The ICU User Guide on regexes lists what follows.

[abc] Match any of the characters a, b, or c
[^abc] Negation – match any character except a, b, or c
[A-M] Range – match any character from A to M (based on Unicode code point ordering)
[\p{L}],[\p{Letter}],[\p{General_Category=Letter}],[[:letter:]] Characters with Unicode Category = Letter (4 equivalent forms)
[^\p{Letter}] Negated property – match everything except Letters
[\p{numeric_value=9}] Match all numbers with a numeric value of 9
[^\p{Letter}&&\p{script=cyrillic}] Intersection; match the set of all Cyrillic letters
[^\p{Letter}--\p{script=latin}] Set difference; match all non-Latin letters
[[a-z][A-Z][0-9]], [a-zA-Z0-9] Union; match ASCII letters and digits (2 equivalent forms)

Regex Functions in stringi

Note that if a given regex pattern is empty, then all the functions in stringi give NA in result and generate a warning. On a syntax error, a quite informative failure message is shown.

If you wish to search for a fixed pattern, refer to about_search_coll or about_search_fixed. They allow to perform a locale-aware text lookup, or a very fast exact-byte search, respectively.

Author(s)

Marek Gagolewski and other contributors

References


J.E.F. Friedl, Mastering Regular Expressions, O’Reilly, 2002

Unicode Regular Expressions – Unicode Technical Standard #18, https://www.unicode.org/reports/tr18/

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_regex: about_search, stri_opts_regex()

Other stringi_general_topics: about_arguments, about_encoding, about_locale, about_search_boundaries, about_search_charclass, about_search_coll, about_search_fixed, about_search, about_stringi

---

**Description**

stringi is THE R package for fast, correct, consistent, and convenient string/text manipulation. It gives predictable results on every platform, in each locale, and under any native character encoding.

**Keywords:** R, text processing, character strings, internationalization, localization, ICU, ICU4C, i18n, l10n, Unicode.

**Homepage:** https://stringi.gagolewski.com/

**License:** The BSD-3-clause license for the package code, the ICU license for the accompanying ICU4C distribution, and the UCD license for the Unicode Character Database. See the COPYRIGHTS and LICENSE file for more details.

**Details**

Manual pages on general topics:

- **about_encoding** – character encoding issues, including information on encoding management in stringi, as well as on encoding detection and conversion.
- **about_locale** – locale issues, including locale management and specification in stringi, and the list of locale-sensitive operations. In particular, see stri_opts_collator for a description of the string collation algorithm, which is used for string comparing, ordering, ranking, sorting, case-folding, and searching.
- **about_arguments** – information on how stringi handles the arguments passed to its function.

**Facilities available**

Refer to the following:

- **about_search** for string searching facilities; these include pattern searching, matching, string splitting, and so on. The following independent search engines are provided:
  - **about_search_regex** – with ICU (Java-like) regular expressions,
  - **about_search_fixed** – fast, locale-independent, byte-wise pattern matching,
  - **about_search_coll** – locale-aware pattern matching for natural language processing tasks,
– **about_search_charclass** – seeking elements of particular character classes, like “all whitespaces” or “all digits”;
– **about_search_boundaries** – text boundary analysis.

- **stri_datetime_format** for date/time formatting and parsing. Also refer to the links therein for other date/time/time zone-related operations.
- **stri_stats_general** and **stri_stats_latex** for gathering some fancy statistics on a character vector’s contents.
- **stri_sub**, **stri_dup**, `%s%`, and **stri_flatten** for concatenation-based operations.
- **stri_length** (among others) for determining the number of code points in a string. See also **stri_count_boundaries** for counting the number of Unicode characters and **stri_width** for approximating the width of a string.
- **stri_trim** (among others) for trimming characters from the beginning or/and end of a string, see also **about_search_charclass**, and **stri_pad** for padding strings so that they are of the same width. Additionally, **stri_wrap** wraps text into lines.
- **stri_trans_tolower** (among others) for case mapping, i.e., conversion to lower, UPPER, or Title Case, **stri_trans_nfc** (among others) for Unicode normalization, **stri_trans_char** for translating individual code points, and **stri_trans_general** for other universal text transforms, including transliteration.
- **stri_cmp**, `%s<%`, **stri_order**, **stri_sort**, **stri_rank**, **stri_unique**, and **stri_duplicated** for collation-based, locale-aware operations, see also **about_locale**.
- **stri_split_lines** (among others) to split a string into text lines.
- **stri_escape_unicode** (among others) for escaping some code points.
- **stri_rand_strings**, **stri_rand_shuffle**, and **stri_rand_lipsum** for generating (pseudo)random strings.
- **stri_read_raw**, **stri_read_lines**, and **stri_write_lines** for reading and writing text files.

Note that each man page provides many further links to other interesting facilities and topics.

**Author(s)**

Marek Gagolewski, with contributions from Bartek Tartanus and many others. ICU4C was developed by IBM, Unicode, Inc., and others.

**References**

**stringi** Package Homepage, [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, [doi:10.18637/jss.v103.i02](https://doi.org/10.18637/jss.v103.i02)

*ICU – International Components for Unicode*, [https://icu.unicode.org/](https://icu.unicode.org/)

*ICU4C API Documentation*, [https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/](https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/)

*The Unicode Consortium*, [https://home.unicode.org/](https://home.unicode.org/)


**stri_compare**

**See Also**

The official online manual of **stringi** at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other stringi_general_topics: **about_arguments**, **about_encoding**, **about_locale**, **about_search_boundaries**, **about_search_charclass**, **about_search_coll**, **about_search_fixed**, **about_search_regex**, **about_search**

---

**stri_compare**                  **Compare Strings with or without Collation**

**Description**

These functions may be used to determine if two strings are equal, canonically equivalent (this is performed in a much more clever fashion than when testing for equality), or to check whether they are in a specific lexicographic order.

**Usage**

```r
stri_compare(e1, e2, ..., opts_collator = NULL)
stri_cmp(e1, e2, ..., opts_collator = NULL)
stri_cmp_eq(e1, e2)
stri_cmp_neq(e1, e2)
stri_cmp_equiv(e1, e2, ..., opts_collator = NULL)
stri_cmp_nequiv(e1, e2, ..., opts_collator = NULL)
stri_cmp_lt(e1, e2, ..., opts_collator = NULL)
stri_cmp_gt(e1, e2, ..., opts_collator = NULL)
stri_cmp_le(e1, e2, ..., opts_collator = NULL)
stri_cmp_ge(e1, e2, ..., opts_collator = NULL)
```

**Arguments**

- `e1, e2` character vectors or objects coercible to character vectors
- `...` additional settings for `opts_collator`
- `opts_collator` a named list with **ICU** Collator's options, see **stri_opts_collator**, NULL for the default collation options.
Details

All the functions listed here are vectorized over e1 and e2.

stri_cmp_eq tests whether two corresponding strings consist of exactly the same code points, while stri_cmp_neq allows to check whether there is any difference between them. These are locale-independent operations: for natural language processing, where the notion of canonical equivalence is more valid, this might not be exactly what you are looking for, see Examples. Please note that stringi always silently removes UTF-8 BOMs from input strings, therefore, e.g., stri_cmp_eq does not take BOMs into account while comparing strings.

stri_cmp_equiv tests for canonical equivalence of two strings and is locale-dependent. Additionally, the ICU’s Collator may be tuned up so that, e.g., the comparison is case-insensitive. To test whether two strings are not canonically equivalent, call stri_cmp_nequiv.

stri_cmp_le tests whether the elements in the first vector are less than or equal to the corresponding elements in the second vector, stri_cmp_ge tests whether they are greater or equal, stri_cmp_lt if less, and stri_cmp_gt if greater, see also, e.g., %s<%.

stri_compare is an alias to stri_cmp. They both perform exactly the same locale-dependent operation. Both functions provide a C library’s strcmp() look-and-feel, see Value for details.

For more information on ICU’s Collator and how to tune its settings refer to stri_opts_collator. Note that different locale settings may lead to different results (see the examples below).

Value

The stri_cmp and stri_compare functions return an integer vector representing the comparison results: -1 if e1[...] < e2[...], 0 if they are canonically equivalent, and 1 if greater.

All the other functions return a logical vector that indicates whether a given relation holds between two corresponding elements in e1 and e2.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other locale_sensitive: %s<%, about_locale, about_search_boundaries, about_search_coll, stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_collator(), stri_order(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()
**Examples**

# in Polish, ch < h:
stri_cmp_lt('hladny', 'chladny', locale='pl_PL')

# in Slovak, ch > h:
stri_cmp_lt('hladny', 'chladny', locale='sk_SK')

# < or > (depends on locale):
stri_cmp('hladny', 'chladny')

# ignore case differences:
stri_cmp_equiv('hladny', 'HLADNY', strength=2)

# also ignore diacritical differences:
stri_cmp_equiv('hladn\u00FD', 'hladny', strength=1, locale='sk_SK')

marios <- c('Mario', 'mario', 'M\u00e1rio', 'm\u00e1rio')
stri_cmp_equiv(marios, 'mario', case_level=TRUE, strength=2L)
stri_cmp_equiv(marios, 'mario', case_level=TRUE, strength=1L)
stri_cmp_equiv(marios, 'mario', strength=1L)
stri_cmp_equiv(marios, 'mario', strength=2L)

# non-Unicode-normalized vs normalized string:
stri_cmp_equiv(stri_trans_nfkd('\u0105'), '\u105')

# note the difference:
stri_cmp_eq(stri_trans_nfkd('\u0105'), '\u105')

# ligatures:
stri_cmp_equiv('\ufb00', 'ff', strength=2)

# phonebook collation
stri_cmp_equiv('G\u00e4rtner', 'Gaertner', locale='de_DE@collation=phonebook', strength=1L)
stri_cmp_equiv('G\u00e4rtner', 'Gaertner', locale='de_DE', strength=1L)

---

**stri_count**

*Count the Number of Pattern Occurrences*

**Description**

These functions count the number of occurrences of a pattern in a string.

**Usage**

stri_count(str, ..., regex, fixed, coll, charclass)

stri_count_charclass(str, pattern)
stri_count_coll(str, pattern, ..., opts_collator = NULL)
stri_count_fixed(str, pattern, ..., opts_fixed = NULL)
stri_count_regex(str, pattern, ..., opts_regex = NULL)

Arguments

str character vector; strings to search in
...
  supplementary arguments passed to the underlying functions, including addi-
tional settings for opts_collator, opts_regex, opts_fixed, and so on
pattern, regex, fixed, coll, charclass
  character vector; search patterns; for more details refer to stringi-search
opts_collator, opts_fixed, opts_regex
  a named list used to tune up the search engine’s settings; see stri_opts_collator,
  stri_opts_fixed, and stri_opts_regex, respectively; NULL for the defaults

Details

Vectorized over str and pattern (with recycling of the elements in the shorter vector if necessary). This allows to, for instance, search for one pattern in each given string, search for each pattern in one given string, and search for the i-th pattern within the i-th string.

If pattern is empty, then the result is NA and a warning is generated.

stri_count is a convenience function. It calls either stri_count_regex, stri_count_fixed, stri_count_coll, or stri_count_charclass, depending on the argument used.

Value

All the functions return an integer vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/
Other search_count: about_search, stri_count_boundaries()

Examples

s <- 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.'
stri_count(s, fixed='dolor')
stri_count(s, regex='\p{L}+')
stri_count_fixed(s, ' ')
**stri_count_boundaries**  

*Count the Number of Text Boundaries*

**Description**

These functions determine the number of text boundaries (like character, word, line, or sentence boundaries) in a string.

**Usage**

```r
stri_count_boundaries(str, ..., opts_brkiter = NULL)
stri_count_words(str, locale = NULL)
```

**Arguments**

- `str` character vector or an object coercible to
- `...` additional settings for `opts_brkiter`
- `opts_brkiter` a named list with ICU BreakIterator’s settings, see `stri_opts_brkiter`; NULL for the default break iterator, i.e., `line_break`
- `locale` NULL or `'` for text boundary analysis following the conventions of the default locale, or a single string with locale identifier, see `stringi-locale`

**Details**

Vectorized over `str`.

For more information on text boundary analysis performed by ICU’s BreakIterator, see `stringi-search-boundaries`.  

```r
stri_count_fixed(s, 'o')
stri_count_fixed(s, 'it')
stri_count_fixed(s, letters)
stri_count_fixed('babab', 'b')
stri_count_fixed(c('stringi', '123'), 'string')

stri_count_charclass(c('stRRRingi', 'STrrrINGI', '123'),
                    c('\p{LL}', '\p{LU}', '\p{Zs}'))
stri_count_charclass(' \t\n', '\p{WHITE_SPACE}') # white space - binary property
stri_count_charclass(' \t\n', '\p{Z}') # white-space - general category (note the difference)

stri_count_regex(s, '(s|el)it')
stri_count_regex(s, 'i.i')
stri_count_regex(s, '.it')
stri_count_regex('bab baab baabab', c('b.*b', 'b.b'))
stri_count_regex(c('stringi', '123'), '^s[1]')
```
In case of `stri_count_words`, just like in `stri_extract_all_words` and `stri_locate_all_words`, ICU’s word BreakIterator iterator is used to locate the word boundaries, and all non-word characters (UBRK_WORD_NONE rule status) are ignored. This function is equivalent to a call to `stri_count_boundaries(str, type='word', skip_word_none=TRUE, locale=locale).

Note that a BreakIterator of type character may be used to count the number of Unicode characters in a string. The `stri_length` function, which aims to count the number of Unicode code points, might report different results.

Moreover, a BreakIterator of type sentence may be used to count the number of sentences in a text piece.

**Value**

Both functions return an integer vector.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other search_count: `about_search`, `stri_count()`

Other locale_sensitive: `%s<%()`, `about_locale`, `about_search_boundaries`, `about_search_coll`, `stri_compare()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_collator()`, `stri_order()`, `stri_rank()`, `stri_sort_key()`, `stri_sort()`, `stri_split_boundaries()`, `stri_trans_tolower()`, `stri_unique()`, `stri_wrap()`

Other text_boundaries: `about_search_boundaries`, `about_search`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_brkiter()`, `stri_split_boundaries()`, `stri_split_lines()`, `stri_trans_tolower()`, `stri_wrap()`

**Examples**

```r
test <- 'The\u00a0above-mentioned features are very useful. Spam, spam, eggs, bacon, and spam.'
stri_count_boundaries(test, type='word')
stri_count_boundaries(test, type='sentence')
stri_count_words(test)

test2 <- stri_trans_nfkd('\u03c0\u0153\u0119\u00a9\u00df\u2190\u2193\u2192')
stri_count_boundaries(test2, type='character')
stri_length(test2)
stri_numbytes(test2)
```
stri_datetime_add  

**Description**

Modifies a date-time object by adding a specific amount of time units.

**Usage**

```r
stri_datetime_add(
  time,
  value = 1L,
  units = "seconds",
  tz = NULL,
  locale = NULL
)
```

```r
stri_datetime_add(time, units = "seconds", tz = NULL, locale = NULL) <- value
```

**Arguments**

- `time` an object of class `POSIXct` (as `as.POSIXct` will be called on character vectors and objects of class `POSIXlt`, `Date`, and `factor`)
- `value` integer vector; signed number of units to add to `time`
- `units` single string; one of 'years', 'months', 'weeks', 'days', 'hours', 'minutes', 'seconds', or 'milliseconds'
- `tz` `NULL` or `''` for the default time zone or a single string with a timezone identifier, `locale` `NULL` or `''` for default locale, or a single string with locale identifier; a non-Gregorian calendar may be specified by setting the `@calendar=name` keyword

**Details**

Vectorized over `time` and `value`.

Note that, e.g., January, 31 + 1 month = February, 28 or 29.

**Value**

Both functions return an object of class `POSIXct`.

The replacement version of `stri_datetime_add` modifies the state of the time object.

**Author(s)**

Marek Gagolewski and other contributors
References


See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other datetime: `stri_datetime_create()`, `stri_datetime_fields()`, `stri_datetime_format()`, `stri_datetime_fstr()`, `stri_datetime_now()`, `stri_datetime_symbols()`, `stri_timezone_get()`, `stri_timezone_info()`, `stri_timezone_list()`

Examples

```r
x <- stri_datetime_now()
stri_datetime_add(x, units='months') <- 2
print(x)
stri_datetime_add(x, -2, units='months')
stri_datetime_add(stri_datetime_create(2014, 4, 20), 1, units='years')
stri_datetime_add(stri_datetime_create(2014, 4, 20), 1, units='years', locale='@calendar=hebrew')

stri_datetime_add(stri_datetime_create(2016, 1, 31), 1, units='months')
```

---

`stri_datetime_create`  *Create a Date-Time Object*

Description

Constructs date-time objects from numeric representations.

Usage

```r
stri_datetime_create(
  year = NULL,
  month = NULL,
  day = NULL,
  hour = 0L,
  minute = 0L,
  second = 0,
  lenient = FALSE,
  tz = NULL,
  locale = NULL
)
```
**Arguments**

- `year` integer vector; 0 is 1BCE, -1 is 2BCE, etc.; NULL for the current year
- `month` integer vector; months are 1-based; NULL for the current month
- `day` integer vector; NULL for the current day
- `hour` integer vector; NULL for the current hour
- `minute` integer vector; NULL for the current minute
- `second` numeric vector; fractional seconds are allowed; NULL for the current seconds (without milliseconds)
- `lenient` single logical value; should the operation be lenient?
- `tz` NULL or '' for the default time zone or a single string with time zone identifier, see `stri_timezone_list`
- `locale` NULL or '' for default locale, or a single string with locale identifier; a non-Gregorian calendar may be specified by setting `@calendar=name` keyword

**Details**

Vectorized over `year`, `month`, `day`, `hour`, `hour`, `minute`, and `second`.

**Value**

Returns an object of class `POSIXct`.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other datetime: `stri_datetime_add()`, `stri_datetime_fields()`, `stri_datetime_format()`, `stri_datetime_fstr()`, `stri_datetime_now()`, `stri_datetime_symbols()`, `stri_timezone_get()`, `stri_timezone_info()`, `stri_timezone_list()`

**Examples**

```r
stri_datetime_create(2015, 12, 31, 23, 59, 59.999)
stri_datetime_create(5775, 8, 1, locale='@calendar=hebrew') # 1 Nisan 5775 -> 2015-03-21
stri_datetime_create(2015, 02, 29)
stri_datetime_create(2015, 02, 29, lenient=TRUE)
stri_datetime_create(hour=15, minute=59)
```
stri_datetime_fields  

Get Values for Date and Time Fields

Description

Computes and returns values for all date and time fields.

Usage

stri_datetime_fields(time, tz = attr(time, "tzone"), locale = NULL)

Arguments

time an object of class POSIXct (as.POSIXct will be called on character vectors and objects of class POSIXlt, Date, and factor)
tz NULL or ' ' for the default time zone or a single string with time zone identifier, see stri_timezone_list
locale NULL or ' ' for the current default locale, or a single string with a locale identifier; a non-Gregorian calendar may be specified by setting @calendar=name keyword

Details

Vectorized over time.

Value

Returns a data frame with the following columns:

1. Year (0 is 1BC, -1 is 2BC, etc.)
2. Month (1-based, i.e., 1 stands for the first month, e.g., January; note that the number of months depends on the selected calendar, see stri_datetime_symbols)
3. Day
4. Hour (24-h clock)
5. Minute
6. Second
7. Millisecond
8. WeekOfYear (this is locale-dependent)
9. WeekOfMonth (this is locale-dependent)
10. DayOfYear
11. DayOfWeek (1-based, 1 denotes Sunday; see stri_datetime_symbols)
12. Hour12 (12-h clock)
13. AmPm (see stri_datetime_symbols)
14. Era (see stri_datetime_symbols)
stri_datetime_format

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other datetime: stri_datetime_add(), stri_datetime_create(), stri_datetime_format(), stri_datetime_fstr(), stri_datetime_now(), stri_datetime_symbols(), stri_timezone_get(), stri_timezone_info(), stri_timezone_list()

Examples

stri_datetime_fields(stri_datetime_now())
stri_datetime_fields(stri_datetime_now(), locale=’@calendar=hebrew’) $Month
stri_datetime_symbols(locale=’@calendar=hebrew’) $Month

stri_datetime_format Date and Time Formatting and Parsing

Description

These functions convert a given date/time object to a character vector, or vice versa.

Usage

stri_datetime_format(  
time,  
format = "uuuu-MM-dd HH:mm:ss",  
tz = NULL,  
locale = NULL  
)

stri_datetime_parse(  
str,  
format = "uuuu-MM-dd HH:mm:ss",  
lenient = FALSE,  
tz = NULL,  
locale = NULL  
)
Arguments

- **time**: an object of class `POSIXct` with date-time data to be formatted (as `POSIXct` will be called on character vectors and objects of class `POSIXlt`, `Date`, and `factor`)
- **format**: character vector, see Details; see also `stri_datetime_fstr`
- **tz**: NULL or '' for the default time zone or a single string with a timezone identifier, see `stri_timezone_get` and `stri_timezone_list`
- **locale**: NULL or '' for the default locale, or a single string with locale identifier; a non-Gregorian calendar may be specified by setting the `@calendar` keyword
- **str**: character vector with strings to be parsed
- **lenient**: single logical value; should date/time parsing be lenient?

Details

Vectorized over `format` and `time` or `str`.

When parsing strings, unspecified date-time fields (e.g., seconds where only hours and minutes are given) are based on today’s midnight in the local time zone (for compatibility with `strptime`).

By default, `stri_datetime_format` (for compatibility with the `strftime` function) formats a date/time object using the current default time zone.

`format` may be one of `DT_STYLE` or `DT_relative_STYLE`, where `DT` is equal to `date`, `time`, or `datetime`, and `STYLE` is equal to `full`, `long`, `medium`, or `short`. This gives a locale-dependent date and/or time format. Note that currently ICU does not support relative time formats, thus this flag is currently ignored in such a context.

Otherwise, `format` is a pattern: a string where specific sequences of characters are replaced with date/time data from a calendar when formatting or used to generate data for a calendar when parsing. For example, `y` stands for 'year'. Characters may be used multiple times: `yy` might produce `99`, whereas `yyyy` yields `1999`. For most numerical fields, the number of characters specifies the field width. For example, if `h` is the hour, `h` might produce `5`, but `hh` yields `05`. For some characters, the count specifies whether an abbreviated or full form should be used.

Two single quotes represent a literal single quote, either inside or outside single quotes. Text within single quotes is not interpreted in any way (except for two adjacent single quotes). Otherwise, all ASCII letters from `a` to `z` and `A` to `Z` are reserved as syntax characters, and require quoting if they are to represent literal characters. In addition, certain ASCII punctuation characters may become available in the future (e.g., `:` being interpreted as the time separator and `/` as a date separator, and replaced by respective locale-sensitive characters in display).

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example(s)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>era designator</td>
<td>G, GG, or GGG</td>
<td>AD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GGGG</td>
<td>Anno Domini</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GGGGG</td>
<td>A</td>
</tr>
<tr>
<td>y</td>
<td>year</td>
<td>yy</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y or yyy</td>
<td>1996</td>
</tr>
<tr>
<td>u</td>
<td>extended year</td>
<td>u</td>
<td>4601</td>
</tr>
<tr>
<td>U</td>
<td>cyclic year name, as in Chinese lunar calendar</td>
<td>U</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>related Gregorian year</td>
<td>r</td>
<td>1996</td>
</tr>
<tr>
<td>Q</td>
<td>quarter</td>
<td>Q or QQ</td>
<td>02</td>
</tr>
</tbody>
</table>
### stri_datetime_format

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>Stand Alone quarter</td>
<td>QQQ, QQQQ, QQQQQ, q, qq, qqq, qqqqq, 02, Q2, 2, 2nd quarter</td>
</tr>
<tr>
<td>M</td>
<td>month in year</td>
<td>M, MM, MMM, MMMM, M or MM, MMMM, MMMMM, 09, Sep, September</td>
</tr>
<tr>
<td>L</td>
<td>Stand Alone month in year</td>
<td>L, LL, LLL, LLLLL, L or LL, LLL, LLLLL, September, S</td>
</tr>
<tr>
<td>w</td>
<td>week of year</td>
<td>w, ww, w or ww, 27</td>
</tr>
<tr>
<td>W</td>
<td>week of month</td>
<td>W, 2</td>
</tr>
<tr>
<td>d</td>
<td>day in month</td>
<td>d, dd, d or dd, 2, 02, 24, 189</td>
</tr>
<tr>
<td>D</td>
<td>day of year</td>
<td>D, dd, D or dd, 189</td>
</tr>
<tr>
<td>F</td>
<td>day of week in month</td>
<td>F, F or g, 2, 02, 02, 2, 2 (2nd Wed in July)</td>
</tr>
<tr>
<td>g</td>
<td>modified Julian day</td>
<td>g, g or F, 2451334</td>
</tr>
<tr>
<td>E</td>
<td>day of week</td>
<td>E, EE, or EEE, EEEE, EEEEE, Tue, Tuesday</td>
</tr>
<tr>
<td>e</td>
<td>local day of week</td>
<td>e, ee, e or ee, eee, eee, eee, eee, eee, eee, eee, eee, Tue, Tuesday</td>
</tr>
<tr>
<td>c</td>
<td>Stand Alone local day of week</td>
<td>c, cc, c or cc, ccc, cccc, cccc, 2</td>
</tr>
<tr>
<td>a</td>
<td>am/pm marker</td>
<td>a, pm</td>
</tr>
<tr>
<td>h</td>
<td>hour in am/pm (1~12)</td>
<td>h, h or hh, 7, 07</td>
</tr>
<tr>
<td>H</td>
<td>hour in day (0~23)</td>
<td>H, HH, H or HH, 0, 00</td>
</tr>
<tr>
<td>k</td>
<td>hour in day (1~24)</td>
<td>k, kk, k or kk, 24, 24</td>
</tr>
<tr>
<td>K</td>
<td>hour in am/pm (0~11)</td>
<td>K, KK, K or KK, 0, 00</td>
</tr>
<tr>
<td>m</td>
<td>minute in hour</td>
<td>m, mm, m or mm, 4, 04</td>
</tr>
<tr>
<td>s</td>
<td>second in minute</td>
<td>s, s or ss, 5</td>
</tr>
</tbody>
</table>
str_datetime_format

S fractional second - truncates (like other time fields) to the count of letters when formatting. Appends zeros if more than 3 letters specified. Truncates at three significant digits when parsing.
A milliseconds in day
z Time Zone: specific non-location

Z Time Zone: ISO8601 basic hms? / RFC 822
Time Zone: long localized GMT (=OOOO)
Time Zone: ISO8601 extended hms? (=XXXXX)
Time Zone: short localized GMT
Time Zone: long localized GMT (=ZZZZZ)
Time Zone: generic non-location (falls back first to VVVVV)
Time Zone: short time zone ID
Time Zone: long time zone ID
Time Zone: time zone exemplar city
Time Zone: generic location (falls back to OOOO)

O Time Zone: ISO8601 basic hms?, with Z for 0
Time Zone: ISO8601 basic hm, with Z
Time Zone: ISO8601 extended hm, with Z
Time Zone: ISO8601 basic hms?, with Z
Time Zone: ISO8601 extended hms?, with Z

V Time Zone: ISO8601 basic hms?, with Z
Time Zone: ISO8601 basic hm, with Z

x Time Zone: ISO8601 basic hms?, without Z for 0
Time Zone: ISO8601 basic hm, without Z
Time Zone: ISO8601 extended hm, without Z
Time Zone: ISO8601 basic hms?, without Z
Time Zone: ISO8601 extended hms?, without Z

Note that any characters in the pattern that are not in the ranges of [a-z] and [A-Z] will be treated as quoted text. For instance, characters like : , ., (a space), # and @ will appear in the resulting time text even if they are not enclosed within single quotes. The single quote is used to “escape” the letters. Two single quotes in a row, inside or outside a quoted sequence, represent a “real” single quote.

A few examples:

<table>
<thead>
<tr>
<th>Example Pattern</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy.MM.dd 'at' HH:mm:ss zzz z</td>
<td>2015.12.31 at 23:59:59 GMT+1</td>
</tr>
<tr>
<td>EEE, MMM d, &quot;yy h:mm a</td>
<td>czw., gru 31, ’15</td>
</tr>
<tr>
<td>hh ’o’clock’ a, zzzz</td>
<td>11 o’clock PM, GMT+01:00</td>
</tr>
<tr>
<td>K:mm a, z</td>
<td>11:59 PM, GMT+1</td>
</tr>
<tr>
<td>yyyyMMdd GGG hh:mm aaaa</td>
<td>2015.grudnia.31 n.e. 11:59 PM</td>
</tr>
<tr>
<td>uuuu-MM-dd’T’HH:mm:ssZ</td>
<td>2015-12-31T23:59:59+0100 (the ISO 8601 guideline)</td>
</tr>
</tbody>
</table>
**stri_datetime_fstr**

Value

stri_datetime_format returns a character vector.
stri_datetime_parse returns an object of class POSIXct.

**Author(s)**

Marek Gagolewski and other contributors

**References**


**See Also**

The official online manual of stringi at https://stringi.gagolewski.com/


Other datetime: stri_datetime_add(), stri_datetime_create(), stri_datetime_fields(), stri_datetime_fstr(), stri_datetime_now(), stri_datetime_symbols(), stri_timezone_get(), stri_timezone_info(), stri_timezone_list()

**Examples**

```r
x <- c("2015-02-28", "2015-02-29")
stri_datetime_parse(x, "yyyy-MM-dd")
stri_datetime_parse(x, "yyyy-MM-dd", lenient=TRUE)
stri_datetime_parse(x %s+" 17:13", "yyyy-MM-dd HH:mm")
stri_datetime_parse("19 lipca 2015", "date_long", locale="pl_PL")
stri_datetime_format(stri_datetime_now(), "datetime_relative_medium")
```

==stri_datetime_fstr==

**Convert strptime-Style Format Strings**

**Description**

This function converts strptime or strftime-style format strings to ICU format strings that may be used in stri_datetime_parse and stri_datetime_format functions.

**Usage**

```r
stri_datetime_fstr(x, ignore_special = TRUE)
```
Arguments

- **x** character vector of date/time format strings
- **ignore_special** if FALSE, special identifiers like "datetime_full" or date_relative_short (see **stri_datetime_format**) are left as-is

Details

For more details on conversion specifiers please refer to the manual page of **strptime**. Most of the formatters of the form %%x, where x is a letter, are supported. Moreover, each %% is replaced with %.

Warnings are given in the case of %%x, %%X, %%u, %%w, %%g, %%G, %%c, %%U, and %%W as in such circumstances either ICU does not support the functionality requested using the string format API or there are some inconsistencies between base R and ICU.

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of **stringi** at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, Journal of Statistical Software 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other datetime: **stri_datetime_add()**, **stri_datetime_create()**, **stri_datetime_fields()**, **stri_datetime_format()**, **stri_datetime_now()**, **stri_datetime_symbols()**, **stri_timezone_get()**, **stri_timezone_info()**, **stri_timezone_list()**

Examples

```r
stri_datetime_fstr('%Y-%m-%d %H:%M:%S')
```

---

**stri_datetime_now** Get Current Date and Time

Description

Returns the current date and time.

Usage

```r
stri_datetime_now()
```
Details

The current date and time in stringi is represented as the (signed) number of seconds since 1970-01-01 00:00:00 UTC. UTC leap seconds are ignored.

Value

Returns an object of class POSIXct.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other datetime: stri_datetime_add(), stri_datetime_create(), stri_datetime_fields(), stri_datetime_format(), stri_datetime_fstr(), stri_datetime_symbols(), stri_timezone_get(), stri_timezone_info(), stri_timezone_list()

stri_datetime_symbols  List Localizable Date-Time Formatting Data

Description

Returns a list of all localizable date-time formatting data, including month and weekday names, localized AM/PM strings, etc.

Usage

stri_datetime_symbols(locale = NULL, context = "standalone", width = "wide")

Arguments

locale NULL or '' for default locale, or a single string with locale identifier
context single string; one of: 'format', 'standalone'
width single string; one of: 'abbreviated', 'wide', 'narrow'

Details

category stands for a selector for date formatting context and width - for date formatting width.
**stri_datetime_symbols**

**Value**

Returns a list with the following named components:

1. Month - month names,
2. Weekday - weekday names,
3. Quarter - quarter names,
4. AmPm - AM/PM names,
5. Era - era names.

**Author(s)**

Marek Gagolewski and other contributors

**References**


*DateFormatSymbols class* – ICU API Documentation, [https://unicode-org.github.io/icu-docs/apidoc/dev/](https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1DateFormatSymbols.html)


**See Also**

The official online manual of stringi at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other datetime: `stri_datetime_add()`, `stri_datetime_create()`, `stri_datetime_fields()`, `stri_datetime_format()`, `stri_datetime_fstr()`, `stri_datetime_now()`, `stri_timezone_get()`, `stri_timezone_info()`, `stri_timezone_list()`

**Examples**

```r
stri_datetime_symbols() # uses the Gregorian calendar in most locales
stri_datetime_symbols('@calendar=hebrew')
stri_datetime_symbols('he_IL@calendar=hebrew')
stri_datetime_symbols('@calendar=islamic')
stri_datetime_symbols('@calendar=persian')
stri_datetime_symbols('@calendar=indian')
stri_datetime_symbols('@calendar=coptic')
stri_datetime_symbols('@calendar=indian')

stri_datetime_symbols('ja_JP_TRADITIONAL') # uses the Japanese calendar by default
stri_datetime_symbols('th_TH_TRADITIONAL') # uses the Buddhist calendar

stri_datetime_symbols('pl_PL', context='format')
stri_datetime_symbols('pl_PL', context='standalone')

stri_datetime_symbols(width='wide')
```
Description

These functions determine, for each string in `str`, if there is at least one match to a corresponding pattern.

Usage

```r
stri_detect(str, ..., regex, fixed, coll, charclass)

stri_detect_fixed(
  str,
  pattern,
  negate = FALSE,
  max_count = -1,
  ...
  opts_fixed = NULL
)

stri_detect_charclass(str, pattern, negate = FALSE, max_count = -1)

stri_detect_coll(
  str,
  pattern,
  negate = FALSE,
  max_count = -1,
  ...
  opts_collator = NULL
)

stri_detect_regex(
  str,
  pattern,
  negate = FALSE,
  max_count = -1,
  ...
  opts_regex = NULL
)
```
Arguments

`str` character vector; strings to search in

`...` supplementary arguments passed to the underlying functions, including additional settings for `opts_collator`, `opts_regex`, `opts_fixed`, and so on

`pattern`, `regex`, `fixed`, `coll`, `charclass` character vector; search patterns; for more details refer to `stringi-search`

`negate` single logical value; whether a no-match to a pattern is rather of interest

`max_count` single integer; allows to stop searching once a given number of occurrences is detected; `-1` (the default) inspects all elements

`opts_collator`, `opts_fixed`, `opts_regex` a named list used to tune up the search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for the defaults

Details

Vectorized over `str` and `pattern` (with recycling of the elements in the shorter vector if necessary). This allows to, for instance, search for one pattern in each given string, search for each pattern in one given string, and search for the `i`-th pattern within the `i`-th string.

If `pattern` is empty, then the result is `NA` and a warning is generated.

`stri_detect` is a convenience function. It calls either `stri_detect_regex`, `stri_detect_fixed`, `stri_detect_coll`, or `stri_detect_charclass`, depending on the argument used.

See also `stri_startswith` and `stri_endswith` for testing whether a string starts or ends with a match to a given pattern. Moreover, see `stri_subset` for a character vector subsetting.

If `max_count` is negative, then all strings are examined. Otherwise, searching terminates once `max_count` matches (or, if `negate` is `TRUE`, no-matches) are detected. The uninspected cases are marked as missing in the return vector. Be aware that, unless `pattern` is a singleton, the elements in `str` might be inspected in a non-consecutive order.

Value

Each function returns a logical vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)  
Other search_detect: `about_search`, `stri_startswith()`
**Examples**

```r
stri_detect_fixed(c('stringi R', 'R STRINGI', '123'), c('i', 'R', '0'))
stri_detect_fixed(c('stringi R', 'R STRINGI', '123'), 'R')

stri_detect_charclass(c('strRRRingi', 'R STRINGI', '123'),
                      c('\p{Ll}', '\p{Lu}', '\p{Zs}'))

stri_detect_regex(c('stringi R', 'R STRINGI', '123'),
                   'R.*')
stri_detect_regex(c('stringi R', 'R STRINGI', '123'),
                   '[[:alpha:]]*?')
stri_detect_regex(c('stringi R', 'R STRINGI', '123'),
                   '[a-zC1]')
stri_detect_regex(c('stringi R', 'R STRINGI', '123'),
                   '( R|RE)')
stri_detect_regex('stringi', 'STRING.', caseInsensitive=TRUE)

stri_detect_regex(c('abc', 'def', '123', 'ghi', '456', '789', 'jkl'),
                   '^\[0-9]+$', max_count=1)
stri_detect_regex(c('abc', 'def', '123', 'ghi', '456', '789', 'jkl'),
                   '^\[0-9]+$', max_count=2)
stri_detect_regex(c('abc', 'def', '123', 'ghi', '456', '789', 'jkl'),
                   '^\[0-9]+$', negate=TRUE, max_count=3)
```

---

**Description**

Duplicates each `str(e1)` string times(e2) times and concatenates the results.

**Usage**

```r
stri_dup(str, times)
```

```r
e1 %s*% e2

e1 %stri*% e2
```

**Arguments**

- `str, e1` a character vector of strings to be duplicated
- `times, e2` an integer vector with the numbers of times to duplicate each string

**Details**

Vectorized over all arguments.

`e1 %s*% e2` and `e1 %stri*% e2` are synonyms for `stri_dup(e1, e2)`

**Value**

Returns a character vector.
Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other join: %s+%(), stri_flatten(), stri_join_list(), stri_join()

Examples

```r
stri_dup('a', 1:5)
stri_dup(c('a', NA, 'ba'), 4)
stri_dup(c('abc', 'pqrst'), c(4, 2))
"a" %s% 5
```

---

**stri_duplicated**

Determine Duplicated Elements

Description

stri_duplicated() determines which strings in a character vector are duplicates of other elements.

stri_duplicated_any() determines if there are any duplicated strings in a character vector.

Usage

```r
stri_duplicated(
  str,
  from_last = FALSE,
  fromLast = from_last,
  ...,
  opts_collator = NULL
)
```

```r
stri_duplicated_any(
  str,
  from_last = FALSE,
  fromLast = from_last,
  ...,
  opts_collator = NULL
)
```
**Arguments**

- **str**: a character vector
- **from_last**: a single logical value; indicates whether search should be performed from the last to the first string
- **fromLast**: [DEPRECATED] alias of from_last
- ... additional settings for opts_collator
- **opts_collator**: a named list with ICU Collator’s options, see `stri_opts_collator`, NULL for default collation options

**Details**

Missing values are regarded as equal.

Unlike `duplicated` and `anyDuplicated`, these functions test for canonical equivalence of strings (and not whether the strings are just byte wise equal) Such operations are locale-dependent. Hence, `stri_duplicated` and `stri_duplicated_any` are significantly slower (but much better suited for natural language processing) than their base R counterparts.

See also `stri_unique` for extracting unique elements.

**Value**

- `stri_duplicated()` returns a logical vector of the same length as `str`. Each of its elements indicates whether a canonically equivalent string was already found in `str`.
- `stri_duplicated_any()` returns a single non-negative integer. Value of 0 indicates that all the elements in `str` are unique. Otherwise, it gives the index of the first non-unique element.

**Author(s)**

Marek Gagolewski and other contributors

**References**


**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Examples

# In the following examples, we have 3 duplicated values,
# 'a' - 2 times, NA - 1 time
stri_duplicated(c('a', 'b', 'a', NA, 'a', NA))
stri_duplicated(c('a', 'b', 'a', NA, 'a', NA), from_last=TRUE)
stri_duplicated_any(c('a', 'b', 'a', NA, 'a', NA))

# compare the results:
stri_duplicated(c('\u0105', stri_trans_nfkd('\u0105')))  
duplicated(c('\u0105', stri_trans_nfkd('\u0105')))  
stri_duplicated(c('gro\u00df', 'GROSS', 'Gro\u00df', 'Gross'), strength=1)  
duplicated(c('gro\u00df', 'GROSS', 'Gro\u00df', 'Gross'))

---

**stri_encode**  
Convert Strings Between Given Encodings

Description

These functions convert strings between encodings. They aim to serve as a more portable and faster replacement for R's own `iconv`.

Usage

`stri_encode(str, from = NULL, to = NULL, to_raw = FALSE)`

`stri_conv(str, from = NULL, to = NULL, to_raw = FALSE)`

Arguments

- **str**: a character vector, a raw vector, or a list of raw vectors to be converted.
- **from**: input encoding: `NULL` or `''` for the default encoding or internal encoding marks’ usage (see Details); otherwise, a single string with encoding name, see `stri_enc_list`.
- **to**: target encoding: `NULL` or `''` for default encoding (see `stri_enc_get`), or a single string with encoding name.
- **to_raw**: a single logical value; indicates whether a list of raw vectors rather than a character vector should be returned.

Details

`stri_conv` is an alias for `stri_encode`.

Refer to `stri_enc_list` for the list of supported encodings and `stringi-encoding` for a general discussion.

If `from` is either missing, `''`, or `NULL`, and if `str` is a character vector then the marked encodings are used (see `stri_enc_mark`) – in such a case bytes-declared strings are disallowed. Otherwise,
i.e., if \texttt{str} is a \texttt{raw}-type vector or a list of \texttt{raw} vectors, we assume that the input encoding is the current default encoding as given by \texttt{stri_enc_get}.

However, if \texttt{from} is given explicitly, the internal encoding declarations are always ignored.

For \texttt{to_raw=}FALSE, the output strings always have the encodings marked according to the target converter used (as specified by \texttt{to}) and the current default Encoding (ASCII, latin1, UTF-8, native, or bytes in all other cases).

Note that some issues might occur if to indicates, e.g., UTF-16 or UTF-32, as the output strings may have embedded NULs. In such cases, please use \texttt{to_raw=}TRUE and consider specifying a byte order marker (BOM) for portability reasons (e.g., set UTF-16 or UTF-32 which automatically adds the BOMs).

Note that \texttt{stri_encode(as.raw(data), 'encodingname')} is a clever substitute for \texttt{rawToChar}.

In the current version of \texttt{stringi}, if an incorrect code point is found on input, it is replaced with the default (for that target encoding) ‘missing/erroneous’ character (with a warning), e.g., the SUBSTITUTE character (U+001A) or the REPLACEMENT one (U+FFFD). Occurrences thereof can be located in the output string to diagnose the problematic sequences, e.g., by calling: \texttt{stri_locate_all_regex(converted_string, '\ufffd\u001a')}.

Because of the way this function is currently implemented, maximal size of a single string to be converted cannot exceed \textasciitilde{}0.67 GB.

\textbf{Value}

If \texttt{to_raw} is FALSE, then a character vector with encoded strings (and appropriate encoding marks) is returned. Otherwise, a list of vectors of type \texttt{raw} is produced.

\textbf{Author(s)}

Marek Gagolewski and other contributors

\textbf{References}


\textbf{See Also}

The official online manual of \texttt{stringi} at \url{https://stringi.gagolewski.com/}

Gagolewski M., \texttt{stringi}: Fast and portable character string processing in R, \textit{Journal of Statistical Software} 103(2), 2022, 1-59, \url{doi:10.18637/jss.v103.i02}

Other encoding conversion: \texttt{about_encoding}, \texttt{stri_enc_fromutf32()}, \texttt{stri_enc_toascii()}, \texttt{stri_enc_tonative()}, \texttt{stri_enc_toutf32()}, \texttt{stri_enc_toutf8()}
Detect Character Set and Language

Description

This function uses the ICU engine to determine the character set, or encoding, of character data in an unknown format.

Usage

\texttt{stri_enc_detect(str, filter_angle_brackets = FALSE)}

Arguments

- \texttt{str}: character vector, a raw vector, or a list of raw vectors
- \texttt{filter_angle_brackets}: logical; If filtering is enabled, text within angle brackets ('<' and '>') will be removed before detection, which will remove most HTML or XML markup.

Details

Vectorized over \texttt{str} and \texttt{filter_angle_brackets}.

For a character vector input, merging all text lines via \texttt{stri_flatten(str, collapse='\n')} might be needed if \texttt{str} has been obtained via a call to \texttt{readLines} and in fact represents an image of a single text file.

This is, at best, an imprecise operation using statistics and heuristics. Because of this, detection works best if you supply at least a few hundred bytes of character data that is mostly in a single language. However, because the detection only looks at a limited amount of the input data, some of the returned character sets may fail to handle all of the input data. Note that in some cases, the language can be determined along with the encoding.

Several different techniques are used for character set detection. For multi-byte encodings, the sequence of bytes is checked for legible patterns. The detected characters are also checked against a list of frequently used characters in that encoding. For single byte encodings, the data is checked against a list of the most commonly occurring three letter groups for each language that can be written using that encoding.

The detection process can be configured to optionally ignore HTML or XML style markup (using ICU’s internal facilities), which can interfere with the detection process by changing the statistics.

This function should most often be used for byte-marked input strings, especially after loading them from text files and before the main conversion with \texttt{stri_encode}. The input encoding is of course not taken into account here, even if marked.

The following table shows all the encodings that can be detected:

<table>
<thead>
<tr>
<th>Character_Set</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF-8</td>
<td>–</td>
</tr>
<tr>
<td>UTF-16BE</td>
<td>–</td>
</tr>
</tbody>
</table>
UTF-16LE –
UTF-32BE –
UTF-32LE –
Shift_JIS Japanese
ISO-2022-CN Simplified Chinese
GB18030 Chinese
Big5 Traditional Chinese
EUC-JP Japanese
EUC-KR Korean
ISO-8859-1 Danish, Dutch, English, French, German, Italian, Norwegian, Portuguese, Swedish
ISO-8859-2 Czech, Hungarian, Polish, Romanian
ISO-8859-5 Russian
ISO-8859-6 Arabic
ISO-8859-7 Greek
ISO-8859-8 Hebrew
ISO-8859-9 Turkish
windows-1250 Czech, Hungarian, Polish, Romanian
windows-1251 Russian
windows-1252 Danish, Dutch, English, French, German, Italian, Norwegian, Portuguese, Swedish
windows-1253 Greek
windows-1254 Turkish
windows-1255 Hebrew
windows-1256 Arabic
KOI8-R Russian
IBM420 Arabic
IBM424 Hebrew

Value

Returns a list of length equal to the length of str. Each list element is a data frame with the following three named vectors representing all the guesses:

- Encoding – string; guessed encodings; NA on failure,
- Language – string; guessed languages; NA if the language could not be determined (e.g., in case of UTF-8),
- Confidence – numeric in [0,1]; the higher the value, the more confidence there is in the match; NA on failure.

The guesses are ordered by decreasing confidence.

Author(s)

Marek Gagolewski and other contributors
References


See Also

The official online manual of *stringi* at https://stringi.gagolewski.com/


Other encoding_detection: about_encoding, stri_enc_detect2(), stri_enc_isascii(), stri_enc_isutf16be(), stri_enc_isutf8()

Examples

```r
## Not run:
## f <- rawToChar(readBin("/quotesingle.Var test.txt/quotesingle.Var", "raw", 100000))
## stri_enc_detect(f)
```

---

**stri_enc_detect2**

[DEPRECATED] Detect Locale-Sensitive Character Encoding

Description

This function tries to detect character encoding in case the language of text is known.

Usage

```r
stri_enc_detect2(str, locale = NULL)
```

Arguments

- `str`: character vector, a raw vector, or a list of raw vectors
- `locale`: NULL or '' for the default locale, or a single string with locale identifier.

Details

Vectorized over `str`.

First, the text is checked whether it is valid UTF-32BE, UTF-32LE, UTF-16BE, UTF-16LE, UTF-8 (as in `stri_enc_detect`, this is roughly inspired by ICU’s i18n/csrucode.cpp) or ASCII.

If `locale` is not NA and the above fails, the text is checked for the number of occurrences of language-specific code points (data provided by the ICU library) converted to all possible 8-bit encodings that fully cover the indicated language. The encoding is selected based on the greatest number of total byte hits.
The guess is of course imprecise, as it is obtained using statistics and heuristics. Because of this, detection works best if you supply at least a few hundred bytes of character data that is in a single language.

If you have no initial guess on the language and encoding, try with `stri_enc_detect` (uses ICU facilities).

**Value**

Just like `stri_enc_detect`, this function returns a list of length equal to the length of `str`. Each list element is a data frame with the following three named components:

- **Encoding** – string; guessed encodings; NA on failure (if and only if `encodings` is empty).
- **Language** – always NA.
- **Confidence** – numeric in [0,1]; the higher the value, the more confidence there is in the match; NA on failure.

The guesses are ordered by decreasing confidence.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other locale_sensitive: `%s<%()``, about_locale, about_search_boundaries, about_search_coll, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_collator(), stri_order(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()``

Other encoding_detection: about_encoding, stri_enc_detect(), stri_enc_isascii(), stri_enc_isutf16be(), stri_enc_isutf8()

---

`stri_enc_fromutf32` **Convert From UTF-32**

**Description**

This function converts integer vectors, representing sequences of UTF-32 code points, to UTF-8 strings.

**Usage**

`stri_enc_fromutf32(vec)`
Argument
vec a list of integer vectors (or objects coercible to such vectors) or NULLs. For
convenience, a single integer vector can also be given.

Details
UTF-32 is a 32-bit encoding where each Unicode code point corresponds to exactly one integer
value.
This function is a vectorized version of intToUtf8. As usual in stringi, it returns character strings
in UTF-8. See stri_enc_toutf32 for a dual operation.
If an ill-defined code point is given, a warning is generated and the corresponding string is set to
NA. Note that 0s are not allowed in vec, as they are used internally to mark the end of a string (in
the C API).
See also stri_encode for decoding arbitrary byte sequences from any given encoding.

Value
Returns a character vector (in UTF-8). NULLs in the input list are converted to NA_character_.

Author(s)
Marek Gagolewski and other contributors

See Also
The official online manual of stringi at https://stringi.gagolewski.com/
Gagolewski M., stringi: Fast and portable character string processing in R, Journal of Statistical
Software 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02
Other encoding_conversion: about_encoding, stri_enc_toascii(), stri_enc_tonative(), stri_enc_toutf32(),
stri_enc_toutf8(), stri_encode()
**stri_enc_isascii**

Check If a Data Stream Is Possibly in ASCII

**Description**

The function checks whether all bytes in a string are <= 127.

**Usage**

`stri_enc_isascii(str)`

**Details**

An error is raised if the provided encoding is unknown to **ICU** (see `stri_enc_list` for more details).

**Value**

Returns a list with the following components:

- **Name.friendly** – friendly encoding name: MIME Name or JAVA Name or **ICU** Canonical Name (the first of provided ones is selected, see below);
- **Name.ICU** – encoding name as identified by **ICU**;
- **Name.*** – other standardized encoding names, e.g., Name.UTR22, Name.IBM, Name.WINDOWS, Name.JAVA, Name.IANA, Name.MIME (some of them may be unavailable for all the encodings);
- **ASCII.subset** – is ASCII a subset of the given encoding?;
- **Unicode.1to1** – for 8-bit encodings only: are all characters translated to exactly one Unicode code point and is the translation scheme reversible?;
- **CharSize.8bit** – is this an 8-bit encoding, i.e., do we have CharSize.min == CharSize.max and CharSize.min == 1?;
- **CharSize.min** – minimal number of bytes used to represent a UChar (in UTF-16, this is not the same as UChar32)
- **CharSize.max** – maximal number of bytes used to represent a UChar (in UTF-16, this is not the same as UChar32, i.e., does not reflect the maximal code point representation size)

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of **stringi** at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other encoding_management: `about_encoding`, `stri_enc_list()`, `stri_enc_mark()`, `stri_enc_set()`
Arguments

str character vector, a raw vector, or a list of raw vectors

Details

This function is independent of the way R marks encodings in character strings (see Encoding and stringi-encoding).

Value

Returns a logical vector. The i-th element indicates whether the i-th string corresponds to a valid ASCII byte sequence.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/
Other encoding_detection: about_encoding, stri_enc_detect2(), stri_enc_detect(), stri_enc_isutf16be(), stri_enc_isutf8()

Examples

stri_enc_isascii(letters[1:3])
stri_enc_isascii(\u0105\u0104)

Description

These functions detect whether a given byte stream is valid UTF-16LE, UTF-16BE, UTF-32LE, or UTF-32BE.

Usage

stri_enc_isutf16be(str)
stri_enc_isutf16le(str)
stri_enc_isutf32be(str)
stri_enc_isutf32le(str)
**stri_enc_isutf8**

**Check If a Data Stream Is Possibly in UTF-8**

**Description**

The function checks whether given sequences of bytes forms a proper UTF-8 string.

**Usage**

    stri_enc_isutf8(str)

**Arguments**

- **str** character vector, a raw vector, or a list of raw vectors

**Details**

These functions are independent of the way R marks encodings in character strings (see [Encoding](https://www.r-project.org/doc/manuals/r-release/R fortune.html) and [stringi-encoding](https://stringi.gagolewski.com/)). Most often, these functions act on raw vectors.

A result of FALSE means that a string is surely not valid UTF-16 or UTF-32. However, false positives are possible.

Also note that a data stream may be sometimes classified as both valid UTF-16LE and UTF-16BE.

**Value**

Returns a logical vector.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of [stringi](https://stringi.gagolewski.com/) at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other encoding_detection: about_encoding, stri_enc_detect2(), stri_enc_detect(), stri_enc_isascii(), stri_enc_isutf8()
Details

FALSE means that a string is certainly not valid UTF-8. However, false positives are possible. For instance, (c4,85) represents (‘a with ogonek’) in UTF-8 as well as (‘A umlaut’, ’Ellipsis’) in WINDOWS-1250. Also note that UTF-8, as well as most 8-bit encodings, extend ASCII (note that stri_enc_isascii implies that stri_enc_isutf8).

However, the longer the sequence, the greater the possibility that the result is indeed in UTF-8 — this is because not all sequences of bytes are valid UTF-8.

This function is independent of the way R marks encodings in character strings (see Encoding and stringi-encoding).

Value

Returns a logical vector. Its i-th element indicates whether the i-th string corresponds to a valid UTF-8 byte sequence.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other encoding_detection: about_encoding, stri_enc_detect2(), stri_enc_detect(), stri_enc_isascii(), stri_enc_isutf16be()

Examples

stri_enc_isutf8(letters[1:3])
stri_enc_isutf8("\u0105\u0104")
stri_enc_isutf8("\u1234\u0222")

stri_enc_list

List Known Character Encodings

Description

Gives the list of encodings that are supported by ICU.

Usage

stri_enc_list(simplify = TRUE)
**stri_enc_mark**

**Arguments**
- **simplify**: single logical value; return a character vector or a list of character vectors?

**Details**
Apart from given encoding identifiers and their aliases, some other specifiers might additionally be available. This is due to the fact that **ICU** tries to normalize converter names. For instance, `'UTF8'` is also valid, see stringi-encoding for more information.

**Value**
If `simplify` is **FALSE**, a list of character vectors is returned. Each list element represents a unique character encoding. The name attribute gives the **ICU** Canonical Name of an encoding family. The elements (character vectors) are its aliases.
If `simplify` is **TRUE** (the default), then the resulting list is coerced to a character vector and sorted, and returned with removed duplicated entries.

**Author(s)**
Marek Gagolewski and other contributors

**See Also**
The official online manual of **stringi** at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)
Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02
Other encoding_management: about_encoding, stri_enc_info(), stri_enc_mark(), stri_enc_set()

**Examples**
stri_enc_list()
stri_enc_list(FALSE)

---

**Description**
Reads declared encodings for each string in a character vector as seen by **stringi**.

**Usage**
stri_enc_mark(str)

**Arguments**
- **str**: character vector or an object coercible to a character vector
Details

According to Encoding, R has a simple encoding marking mechanism: strings can be declared to be in latin1, UTF-8 or bytes.

Moreover, we may check (via the R/C API) whether a string is in ASCII (R assumes that this holds if and only if all bytes in a string are not greater than 127, so there is an implicit assumption that your platform uses an encoding that extends ASCII) or in the system’s default (a.k.a. unknown in Encoding) encoding.

Intuitively, the default encoding should be equivalent to the one you use on stdin (e.g., your 'keyboard'). In stringi we assume that such an encoding is equivalent to the one returned by stri_enc_get. It is automatically detected by ICU to match – by default – the encoding part of the LC_CTYPE category as given by Sys.getlocale.

Value

Returns a character vector of the same length as str. Unlike in the Encoding function, here the possible encodings are: ASCII, latin1, bytes, native, and UTF-8. Additionally, missing values are handled properly.

This gives exactly the same data that is used by all the functions in stringi to re-encode their inputs.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/

Other encoding_management: about_encoding, stri_enc_info(), stri_enc_list(), stri_enc_set()

stri_enc_set

Set or Get Default Character Encoding in stringi

Description

stri_enc_set sets the encoding used to re-encode strings internally (i.e., by R) declared to be in native encoding, see stringi-encoding and stri_enc_mark. stri_enc_get returns the currently used default encoding.

Usage

stri_enc_set(enc)

stri_enc_get()
**stri_enc_toascii**

Arguments

enc  single string; character encoding name, see *stri_enc_list* for the list of supported encodings.

Details

*stri_enc_get* is the same as *stri_enc_info*(NULL)*$Name.friendly*.

Note that changing the default encoding may have undesired consequences. Unless you are an expert user and you know what you are doing, *stri_enc_set* should only be used if *ICU* fails to detect your system’s encoding correctly (while testing *stringi* we only encountered such a situation on a very old Solaris machine). Note that *ICU* tries to match the encoding part of the *LC_CTYPE* category as given by *Sys.getlocale*.

If you set a default encoding that is neither a superset of ASCII, nor an 8-bit encoding, a warning will be generated, see *stringi-encoding* for discussion.

*stri_enc_set* has no effect if the system *ICU* assumes that the default charset is always UTF-8 (i.e., where the internal U_CHARSET_IS_UTF8 is defined and set to 1), see *stri_info*.

Value

*stri_enc_set* returns a string with previously used character encoding, invisibly.

*stri_enc_get* returns a string with current default character encoding.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of *stringi* at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other encoding_management: *about_encoding*, *stri_enc_info()*, *stri_enc_list()*, *stri_enc_mark()*

---

**Description**

This function converts input strings to ASCII, i.e., to character strings consisting of bytes not greater than 127.

**Usage**

*stri_enc_toascii*(str)
**stri_enc_tonative**

Convert Strings To Native Encoding

**Description**

Converts character strings with declared encodings to the current native encoding.

**Usage**

`stri_enc_tonative(str)`

**Arguments**

- **str** a character vector to be converted
Details
This function just calls `stri_encode(str, NULL, NULL)`. The current native encoding can be read with `stri_enc_get`. Character strings declared to be in bytes encoding will fail here.

Note that if working in a UTF-8 environment, resulting strings will be marked with UTF-8 and not native, see `stri_enc_mark`.

Value
Returns a character vector.

Author(s)
Marek Gagolewski and other contributors

See Also
The official online manual of stringi at https://stringi.gagolewski.com/

Other encoding_conversion: about_encoding, stri_enc_fromutf32(), stri_enc_toascii(), stri_enc_toutf32(), stri_enc_toutf8(), stri_encode()
Description

Converts character strings with declared marked encodings to UTF-8 strings.

Usage

stri_enc_toutf8(str, is_unknown_8bit = FALSE, validate = FALSE)

Arguments

str  a character vector to be converted
is_unknown_8bit  a single logical value, see Details
validate  a single logical value (can be NA), see Details

Details

If is_unknown_8bit is set to FALSE (the default), then R encoding marks are used, see stri_enc_mark. Bytes-marked strings will cause the function to fail.

If a string is in UTF-8 and has a byte order mark (BOM), then the BOM will be silently removed from the output string.

If the default encoding is UTF-8, see stri_enc_get, then strings marked with native are – for efficiency reasons – returned as-is, i.e., with unchanged markings. A similar behavior is observed when calling enc2utf8.

For is_unknown_8bit=TRUE, if a string is declared to be neither in ASCII nor in UTF-8, then all byte codes > 127 are replaced with the Unicode REPLACEMENT CHARACTER (\ufffd). Note that the REPLACEMENT CHARACTER may be interpreted as Unicode missing value for single
stri_escape_unicode

characters. Here a bytes-marked string is assumed to use an 8-bit encoding that extends the ASCII map.
What is more, setting validate to TRUE or NA in both cases validates the resulting UTF-8 byte stream. If validate=TRUE, then in case of any incorrect byte sequences, they will be replaced with the REPLACEMENT CHARACTER. This option may be used in a case where you want to fix an invalid UTF-8 byte sequence. For NA, a bogus string will be replaced with a missing value.

Value
Returns a character vector.

Author(s)
Marek Gagolewski and other contributors

See Also
The official online manual of stringi at https://stringi.gagolewski.com/
Other encoding_conversion: about_encoding,stri_enc_fromutf32(), stri_enc_toascii(), stri_enc_tonative(),stri_enc_toutf32(),stri_encode()

stri_escape_unicode     Escape Unicode Code Points

Description
Generates an ASCII string where all non-printable characters and non-ASCII characters are converted to escape sequences.

Usage
stri_escape_unicode(str)

Arguments
   str          character vector

Details
For non-printable and certain special (well-known, see also the R man page Quotes) ASCII characters, the following (also recognized in R) convention is used. We get \a, \b, \t, \n, \v, \f, \r, \". \' \ or either \uXXXX (4 hex digits) or \UXXXXXXXX (8 hex digits) otherwise.
As usual in stringi, any input string is converted to Unicode before executing the escape process.
Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other escape: stri_unescape_unicode()

Examples

stri_escape_unicode('a\u0105!')

stri_extract_all Extract Pattern Occurrences

Description

These functions extract all substrings matching a given pattern.

stri_extract_all_* extracts all the matches. stri_extract_first_* and stri_extract_last_* yield the first or the last matches, respectively.

Usage

stri_extract_all(str, ..., regex, fixed, coll, charclass)

stri_extract_first(str, ..., regex, fixed, coll, charclass)

stri_extract_last(str, ..., regex, fixed, coll, charclass)

stri_extract(
    str,
    ...,
    regex,
    fixed,
    coll,
    charclass,
    mode = c("first", "all", "last")
)
stri_extract_all_charclass(
    str,
    pattern,
    merge = TRUE,
    simplify = FALSE,
    omit_no_match = FALSE
)

stri_extract_first_charclass(str, pattern)
stri_extract_last_charclass(str, pattern)

stri_extract_all_coll(
    str,
    pattern,
    simplify = FALSE,
    omit_no_match = FALSE,
    ...,
    opts_collator = NULL
)

stri_extract_first_coll(str, pattern, ..., opts_collator = NULL)
stri_extract_last_coll(str, pattern, ..., opts_collator = NULL)

stri_extract_all_regex(
    str,
    pattern,
    simplify = FALSE,
    omit_no_match = FALSE,
    ...,
    opts_regex = NULL
)

stri_extract_first_regex(str, pattern, ..., opts_regex = NULL)
stri_extract_last_regex(str, pattern, ..., opts_regex = NULL)

stri_extract_all_fixed(
    str,
    pattern,
    simplify = FALSE,
    omit_no_match = FALSE,
    ...,
    opts_fixed = NULL
)

stri_extract_first_fixed(str, pattern, ..., opts_fixed = NULL)
stri_extract_last_fixed(str, pattern, ..., opts_fixed = NULL)

Arguments

str character vector; strings to search in

... supplementary arguments passed to the underlying functions, including additional settings for opts_collator, opts_regex, and so on

mode single string; one of: 'first' (the default), 'all', 'last'

pattern, regex, fixed, coll, charclass character vector; search patterns; for more details refer to stringi-search

merge single logical value; indicates whether consecutive pattern matches will be merged into one string; stri_extract_all_charclass only

simplify single logical value; if TRUE or NA, then a character matrix is returned; otherwise (the default), a list of character vectors is given, see Value; stri_extract_all_* only

omit_no_match single logical value; if FALSE, then a missing value will indicate that there was no match; stri_extract_all_* only

opts_collator, opts_fixed, opts_regex a named list to tune up the search engine's settings; see stri_opts_collator, stri_opts_fixed, and stri_opts_regex, respectively; NULL for the defaults

Details

Vectorized over str and pattern (with recycling of the elements in the shorter vector if necessary). This allows to, for instance, search for one pattern in each given string, search for each pattern in one given string, and search for the i-th pattern within the i-th string.

Check out stri_match for the extraction of matches to individual regex capture groups.

stri_extract, stri_extract_all, stri_extract_first, and stri_extract_last are convenience functions. They merely call stri_extract_*_*, depending on the arguments used.

Value

For stri_extract_all*, if simplify=FALSE (the default), then a list of character vectors is returned. Each list element represents the results of a different search scenario. If a pattern is not found and omit_no_match=FALSE, then a character vector of length 1 with single NA value will be generated.

Otherwise, i.e., if simplify is not FALSE, then stri_list2matrix with byrow=TRUE argument is called on the resulting object. In such a case, the function yields a character matrix with an appropriate number of rows (according to the length of str, pattern, etc.). Note that stri_list2matrix's fill argument is set either to an empty string or NA, depending on whether simplify is TRUE or NA, respectively.

stri_extract_first* and stri_extract_last* return a character vector. A NA element indicates a no-match.

Note that stri_extract_last_regex searches from start to end, but skips overlapping matches, see the example below.
stri_extract_all

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_extract: about_search, stri_extract_all_boundaries(), stri_match_all()

Examples

stri_extract_all('XaaaX', regex=c('\p{Ll}', '\p{Ll}+', '\p{Ll}(2,3)', '\p{Ll}(2,3)?'))
stri_extract_all('Bartolini', coll='i')
stri_extract_all('stringi is so good!', charclass='\p{Zs}') # all white-spaces

stri_extract_all_charclass(c('AbcdeFgHiJK', 'abc', 'ABC'), '\p{Ll}')
stri_extract_all_charclass(c('AbcdeFgHiJK', 'abc', 'ABC'), '\p{Ll}', merge=FALSE)
stri_extract_first_charclass('AaBbCc', '\p{Ll}')
stri_extract_last_charclass('AaBbCc', '\p{Ll}')

## Not run:
# emoji support available since ICU 57
stri_extract_all_charclass(stri_enc_fromutf32(32:55200), '\p{EMOJI}')

## End(Not run)

stri_extract_all_coll(c('AaaaaaaaaA', 'AAAA'), 'a')
stri_extract_first_coll(c('Yy\u00FD', 'AAA'), 'y', strength=2, locale='sk_SK')
stri_extract_last_coll(c('Yy\u00FD', 'AAA'), 'y', strength=1, locale='sk_SK')

stri_extract_all_regex('XaaaX', c('\p{Ll}', '\p{Ll}+', '\p{Ll}(2,3)', '\p{Ll}(2,3)?'))
stri_extract_first_regex('XaaaX', c('\p{Ll}', '\p{Ll}+', '\p{Ll}(2,3)', '\p{Ll}(2,3)?'))
stri_extract_last_regex('XaaaX', c('\p{Ll}', '\p{Ll}+', '\p{Ll}(2,3)', '\p{Ll}(2,3)?'))

stri_list2matrix(stri_extract_all_regex('XaaaX', c('\p{Ll}', '\p{Ll}')))  # all white-spaces
stri_extract_all_regex('XaaaX', c('\p{Ll}', '\p{Ll}+'), simplify=TRUE)
stri_extract_all_regex('XaaaX', c('\p{Ll}', '\p{Ll}+'), simplify=NA)

stri_extract_all_fixed('abaBaba', 'Aba', case_insensitive=TRUE)
stri_extract_all_fixed('abaBaba', 'Aba', case_insensitive=TRUE, overlap=TRUE)

# Searching for the last occurrence:
# Note the difference - regex searches left to right, with no overlaps.
stri_extract_last_fixed('agAGA', 'aga', case_insensitive=TRUE)
stri_extract_last_regex('agAGA', 'aga', case_insensitive=TRUE)
stri_extract_all_boundaries

Extract Data Between Text Boundaries

Description

These functions extract data between text boundaries.

Usage

stri_extract_all_boundaries(
  str,
  simplify = FALSE,
  omit_no_match = FALSE,
  ..., 
  opts_brkiter = NULL
)

stri_extract_last_boundaries(str, ..., opts_brkiter = NULL)

stri_extract_first_boundaries(str, ..., opts_brkiter = NULL)

stri_extract_all_words(
  str,
  simplify = FALSE,
  omit_no_match = FALSE,
  locale = NULL
)

stri_extract_first_words(str, locale = NULL)

stri_extract_last_words(str, locale = NULL)

Arguments

str character vector or an object coercible to
simplify single logical value; if TRUE or NA, then a character matrix is returned; otherwise (the default), a list of character vectors is given, see Value
omit_no_match single logical value; if FALSE, then a missing value will indicate that there are no words
... additional settings for opts_brkiter
opts_brkiter a named list with ICU BreakIterator’s settings, see stri_opts_brkiter: NULL for the default break iterator, i.e., line_break
locale NULL or '' for text boundary analysis following the conventions of the default locale, or a single string with locale identifier, see stringi-locale
Details

Vectorized over str.

For more information on text boundary analysis performed by ICU’s BreakIterator, see stringi-search-boundaries.

In case of stri_extract_*_words, just like in stri_count_words, ICU’s word BreakIterator iterator is used to locate the word boundaries, and all non-word characters (UBRK_WORD_NONE rule status) are ignored.

Value

For stri_extract_all_*, if simplify=FALSE (the default), then a list of character vectors is returned. Each string consists of a separate word. In case of omit_no_match=FALSE and if there are no words or if a string is missing, a single NA is provided on output.

Otherwise, stri_list2matrix with byrow=TRUE argument is called on the resulting object. In such a case, a character matrix with length(str) rows is returned. Note that stri_list2matrix’s fill argument is set to an empty string and NA, for simplify TRUE and NA, respectively.

For stri_extract_first_* and stri_extract_last_*, a character vector is returned. A NA element indicates a no-match.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_extract: about_search, stri_extract_all(), stri_match_all()

Other locale_sensitive: %<%( ), about_locale, about_search_boundaries, about_search_coll, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_locate_all_boundaries(), stri_opts_collator(), stri_order(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()

Other text_boundaries: about_search_boundaries, about_search, stri_count_boundaries(), stri_locate_all_boundaries(), stri_opts_brkiter(), stri_split_boundaries(), stri_split_lines(), stri_trans_tolower(), stri_wrap()

Examples

stri_extract_all_words('stringi: THE string processing package 123.48...')
**stri_flatten**  
*Flatten a String*

**Description**

Joins the elements of a character vector into one string.

**Usage**

```r
stri_flatten(str, collapse = "", na_empty = FALSE, omit_empty = FALSE)
```

**Arguments**

- `str`: a vector of strings to be coerced to character
- `collapse`: a single string denoting the separator
- `na_empty`: single logical value; should missing values in `str` be treated as empty strings (TRUE) or be omitted whatsoever (NA)?
- `omit_empty`: single logical value; should empty strings in `str` be omitted?

**Details**

The `stri_flatten(str, collapse='XXX')` call is equivalent to `paste(str, collapse='XXX', sep='')`.

If you wish to use some more fancy (e.g., differing) separators between flattened strings, call `stri_join(str, separators, collapse='')`.

If `str` is not empty, then a single string is returned. If `collapse` has length > 1, then only the first string will be used.

**Value**

Returns a single string, i.e., a character vector of length 1.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other join: `%s+%()`, `stri_dup()`, `stri_join_list()`,
Examples

```r
stri_flatten(LETTERS)
stri_flatten(LETTERS, collapse=''
stri_flatten(stri_dup(letters[1:6], 1:3))
stri_flatten(c(NA, 'A', 'B', NA, 'C'), collapse=''
stri_flatten(c(NA, '', 'A', '', 'B', NA, 'C'), collapse=''
```

### stri_info

**Query Default Settings for stringi**

**Description**

Gives the current default settings used by the ICU library.

**Usage**

```r
stri_info(short = FALSE)
```

**Arguments**

- `short`: logical; whether or not the results should be given in a concise form; defaults to `TRUE`

**Value**

If `short` is `TRUE`, then a single string providing information on the default character encoding, locale, and Unicode as well as ICU version is returned.

Otherwise, a list with the following components is returned:

- `Unicode.version`: version of Unicode supported by the ICU library;
- `ICU.version`: ICU library version used;
- `Locale`: contains information on default locale, as returned by `stri_locale_info`;
- `Charset.internal`: fixed at `c('UTF-8', 'UTF-16')`;
- `Charset.native`: information on the default encoding, as returned by `stri_enc_info`;
- `ICU.system`: logical; `TRUE` indicates that the system ICU libs are used, otherwise ICU was built together with stringi;
- `ICU.UTF8`: logical; `TRUE` if the internal U_CHARSET_IS_UTF8 flag is defined and set.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of stringi at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02
Determine if a String is of Length Zero

Description

This is the fastest way to find out whether the elements of a character vector are empty strings.

Usage

stri_isempty(str)

Arguments

str  character vector or an object coercible to

Details

Missing values are handled properly.

Value

Returns a logical vector of the same length as str.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other length: %s%%(), stri_length(), stri_numbytes(), stri_pad_both(), stri_sprintf(), stri_width()

Examples

stri_isempty(letters[1:3])
stri_isempty(c('','','abc','123','\w0105\w0104'))
stri_isempty(character(1))
Description

These are the stringi's equivalents of the built-in paste function. stri_c and stri_paste are aliases for stri_join.

Usage

stri_join(..., sep = "", collapse = NULL, ignore_null = FALSE)
stri_c(..., sep = "", collapse = NULL, ignore_null = FALSE)
stri_paste(..., sep = "", collapse = NULL, ignore_null = FALSE)

Arguments

... character vectors (or objects coercible to character vectors) whose corresponding elements are to be concatenated
sep a single string; separates terms
collapse a single string or NULL; an optional results separator
ignore_null a single logical value; if TRUE, then empty vectors provided via ... are silently ignored

Details

Vectorized over each atomic vector in '...'.

Unless collapse is NULL, the result will be a single string. Otherwise, you get a character vector of length equal to the length of the longest argument.

If any of the arguments in '...' is a vector of length 0 (not to be confused with vectors of empty strings) and ignore_null is FALSE, then you will get a 0-length character vector in result.

If collapse or sep has length greater than 1, then only the first string will be used.

In case where there are missing values in any of the input vectors, NA is set to the corresponding element. Note that this behavior is different from paste, which treats missing values as ordinary strings like 'NA'. Moreover, as usual in stringi, the resulting strings are always in UTF-8.

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors
stri_join_list

Concatenate Strings in a List

Description

These functions concatenate all the strings in each character vector in a given list. stri_c_list and stri_paste_list are aliases for stri_join_list.

Usage

stri_join_list(x, sep = "", collapse = NULL)
stri_c_list(x, sep = "", collapse = NULL)
stri_paste_list(x, sep = "", collapse = NULL)

Arguments

x a list consisting of character vectors
sep a single string; separates strings in each of the character vectors in x
collapse a single string or NULL; an optional results separator

Details

Unless collapse is NULL, the result will be a single string. Otherwise, you get a character vector of length equal to the length of x.
Vectors in x of length 0 are silently ignored.
If collapse or sep has length greater than 1, then only the first string will be used.
stri_length

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other join: %s+%(). stri_dup(), stri_flatten(), stri_join()

Examples

stri_join_list(
  stri_extract_all_words(c('Lorem ipsum dolor sit amet.',
                           'Spam spam bacon sausage and spam.'),
                           sep=' ', ')
)

stri_join_list(
  stri_extract_all_words(c('Lorem ipsum dolor sit amet.',
                           'Spam spam bacon sausage and spam.'),
                           sep=' ', ', collapse='.' )
)

stri_join_list(
  stri_extract_all_regex(
    c('spam spam bacon', '123 456', 'spam 789 sausage'), '\p{L}+',
  ),
  sep=' ', ')
)

stri_join_list(
  stri_extract_all_regex(
    c('spam spam bacon', '123 456', 'spam 789 sausage'), '\p{L}+',
    omit_no_match=TRUE
  ),
  sep=' ', collapse='; ')

stri_length

Count the Number of Code Points

Description

This function returns the number of code points in each string.
Usage

stri_length(str)

Arguments

str character vector or an object coercible to

Details

Note that the number of code points is not the same as the ‘width’ of the string when printed on the
console.

If a given string is in UTF-8 and has not been properly normalized (e.g., by stri_trans_nfc),
the returned counts may sometimes be misleading. See stri_count_boundaries for a method
to count Unicode characters. Moreover, if an incorrect UTF-8 byte sequence is detected, then a
warning is generated and the corresponding output element is set to NA, see also stri_enc_toutf8
for a method to deal with such cases.

Missing values are handled properly. For ‘byte’ encodings we get, as usual, an error.

Value

Returns an integer vector of the same length as str.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/

Gagolewski M., stringi: Fast and portable character string processing in R, Journal of Statistical
Software 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other length: %s$%(), stri_isempty(), stri_numbytes(), stri_pad_both(), stri_sprintf(),
stri_width()

Examples

stri_length(LETTERS)
stri_length(c('abc', '123', '\u0105\u0104'))
stri_length('\u0105') # length is one, but...
stri_numbytes('\u0105') # 2 bytes are used
stri_numbytes(stri_trans_nfkd('\u0105')) # 3 bytes here but...
stri_length(stri_trans_nfkd('\u0105')) # ...two code points (!)
stri_count_boundaries(stri_trans_nfkd('\u0105'), type='character') # ...and one Unicode character
stri_list2matrix  

Convert a List to a Character Matrix

Description

This function converts a given list of atomic vectors to a character matrix.

Usage

stri_list2matrix(
  x,
  byrow = FALSE,
  fill = NA_character_,
  n_min = 0,
  by_row = byrow
)

Arguments

x  a list of atomic vectors
byrow  a single logical value; should the resulting matrix be transposed?
fill  a single string, see Details
n_min  a single integer value; minimal number of rows (byrow==FALSE) or columns (otherwise) in the resulting matrix
by_row  alias of byrow

Details

This function is similar to the built-in `simplify2array` function. However, it always returns a character matrix, even if each element in x is of length 1 or if elements in x are not of the same lengths. Moreover, the elements in x are always coerced to character vectors.

If byrow is FALSE, then a matrix with `length(x)` columns is returned. The number of rows is the length of the longest vector in x, but no less than n_min. Basically, we have result[i,j] == x[[j]][i] if i <= length(x[[j]]) and result[i,j] == fill otherwise, see Examples.

If byrow is TRUE, then the resulting matrix is a transposition of the above-described one.

This function may be useful, e.g., in connection with `stri_split` and `stri_extract_all`.

Value

Returns a character matrix.

Author(s)

Marek Gagolewski and other contributors
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other utils: stri_na2empty(), stri_remove_empty(), stri_replace_na()

Examples

simplify2array(list(c(`Var`a, `Var`b), c(`Var`c, `Var`d), c(`Var`e, `Var`f)))
stri_list2matrix(list(c(`Var`a, `Var`b), c(`Var`c, `Var`d), c(`Var`e, `Var`f)), byrow=TRUE)

simplify2array(list('a', c('b', 'c')))
stri_list2matrix(list('a', c('b', 'c')))
stri_list2matrix(list('a', c('b', 'c')), fill='')
stri_list2matrix(list('a', c('b', 'c')), fill='', n_min=5)

---

**stri_locale_info**

*Query Given Locale*

**Description**

Provides some basic information on a given locale identifier.

**Usage**

```r
stri_locale_info(locale = NULL)
```

**Arguments**

- `locale` NULL or '' for default locale, or a single string with locale identifier.

**Details**

With this function you may obtain some basic information on any provided locale identifier, even if it is unsupported by ICU or if you pass a malformed locale identifier (the one that is not, e.g., of the form Language_Country). See stringi-locale for discussion.

This function does not do anything really complicated. In many cases it is similar to a call to as.list(stri_split_fixed(locale, '\_', 3L)[[1]]), with locale case mapped. It may be used, however, to get insight on how ICU understands a given locale identifier.

**Value**

Returns a list with the following named character strings: Language, Country, Variant, and Name, being their underscore separated combination.
Description

Creates a character vector with all available locale identifies.

Usage

```r
stri_locale_list()
```

Details

Note that some of the services may be unavailable in some locales. Querying for locale-specific services is always performed during the resource request.

See `stringi-locale` for more information.

Value

Returns a character vector with locale identifiers that are known to ICU.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other locale management: `about_locale, stri_locale_list(), stri_locale_set()`
Examples

```r
stri_locale_list()
```

---

**stri_locale_set**  
*Set or Get Default Locale in stringi*

---

**Description**

`scales_stringi` changes the default locale for all the functions in the *stringi* package, i.e., establishes the meaning of the “NULL locale” argument of locale-sensitive functions. `stri_locale_get` gives the current default locale.

**Usage**

```r
stri_locale_set(locale)
stri_locale_get()
```

**Arguments**

- `locale` single string of the form Language, Language_Country, or Language_Country_Variant, e.g., 'en_US', see `stri_locale_list`.

**Details**

See *stringi-locale* for more information on the effect of changing the default locale.

`scales_stringi_set` is the same as `stri_locale_info(NULL)$Name`.

**Value**

- `stri_locale_set` returns a string with previously used locale, invisibly.
- `stri_locale_get` returns a string of the form Language, Language_Country, or Language_Country_Variant, e.g., 'en_US'.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of *stringi* at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other locale_management: `about_locale`, `stri_locale_info()`, `stri_locale_list()`
Examples

```r
## Not run:
oldloc <- stri_locale_set('pt_BR')
# ... some locale-dependent operations
# ... note that you may always modify a locale per-call
# ... changing the default locale is convenient if you perform
# ... many operations
stri_locale_set(oldloc) # restore the previous default locale

## End(Not run)
```

### stri_locate_all

**Locate Pattern Occurrences**

**Description**

These functions find the indexes (positions) where there is a match to some pattern. The functions `stri_locate_all_*` locate all the matches. `stri_locate_first_*` and `stri_locate_last_*` give the first and the last matches, respectively.

**Usage**

- `stri_locate_all(str, ..., regex, fixed, coll, charclass)`
- `stri_locate_first(str, ..., regex, fixed, coll, charclass)`
- `stri_locate_last(str, ..., regex, fixed, coll, charclass)`
- `stri_locate(str, ..., regex, fixed, coll, charclass, mode = c("first", "all", "last")`)
- `stri_locate_all_charclass(str, pattern, merge = TRUE, omit_no_match = FALSE, get_length = FALSE)`
- `stri_locate_first_charclass(str, pattern, get_length = FALSE)`
stri_locate_last_charclass(str, pattern, get_length = FALSE)

stri_locate_all_coll(
  str,
  pattern,
  omit_no_match = FALSE,
  get_length = FALSE,
  ..., 
  opts_collator = NULL
)

stri_locate_first_coll(
  str,
  pattern,
  get_length = FALSE,
  ..., 
  opts_collator = NULL
)

stri_locate_last_coll(
  str,
  pattern,
  get_length = FALSE,
  ..., 
  opts_collator = NULL
)

stri_locate_all_regex(
  str,
  pattern,
  omit_no_match = FALSE,
  capture_groups = FALSE,
  get_length = FALSE,
  ..., 
  opts_regex = NULL
)

stri_locate_first_regex(
  str,
  pattern,
  capture_groups = FALSE,
  get_length = FALSE,
  ..., 
  opts_regex = NULL
)

stri_locate_last_regex(
Arguments

- **str**: character vector; strings to search in
- **...**: supplementary arguments passed to the underlying functions, including additional settings for `opts_collator`, `opts_regex`, `opts_fixed`, and so on
- **mode**: single string; one of: 'first' (the default), 'all', 'last'
- **pattern, regex, fixed, coll, charclass**: character vector; search patterns; for more details refer to `stringi-search`
- **merge**: single logical value; indicates whether consecutive sequences of indexes in the resulting matrix should be merged; `stri_locate_all_charclass` only
- **omit_no_match**: single logical value; if TRUE, a no-match will be indicated by a matrix with 0 rows `stri_locate_all_*` only
- **get_length**: single logical value; if FALSE (default), generate `from-to` matrices; otherwise, output `from-length` ones
opts_collator, opts_fixed, opts_regex
   named list used to tune up the selected search engine's settings; see stri_opts_collator,
   stri_opts_fixed, and stri_opts_regex, respectively; NULL for the defaults

capture_groups
   single logical value; whether positions of matches to parenthesized subexpressions
   should be returned too (as capture_groups attribute); stri_locate_*_regex only

Details

Vectorized over str and pattern (with recycling of the elements in the shorter vector if necessary).
This allows to, for instance, search for one pattern in each string, search for each pattern in one
string, and search for the i-th pattern within the i-th string.

The matches may be extracted by calling stri_sub or stri_sub_all. Alternatively, you may call
stri_extract directly.

stri_locate, stri_locate_all, stri_locate_first, and stri_locate_last are convenience
functions. They just call stri_locate_*_*, depending on the arguments used.

Value

For stri_locate_all_*, a list of integer matrices is returned. Each list element represents the
results of a separate search scenario. The first column gives the start positions of the matches, and
the second column gives the end positions. Moreover, two NAs in a row denote NA arguments or a
no-match (the latter only if omit_no_match is FALSE).

stri_locate_first_* and stri_locate_last_* return an integer matrix with two columns, giving
the start and end positions of the first or the last matches, respectively, and two NAs if and only
if they are not found.

For stri_locate_*_regex, if the match is of zero length, end will be one character less than start.
Note that stri_locate_last_regex searches from start to end, but skips overlapping matches, see
the example below.

Setting get_length=TRUE results in the 2nd column representing the length of the match instead of
the end position. In this case, negative length denotes a no-match.

If capture_groups=TRUE, then the outputs are equipped with the capture_groups attribute, which
is a list of matrices giving the start-end positions of matches to parenthesized subexpressions. Similarly to stri_match_regex, capture group names are extracted unless looking for first/last occurrences of many different patterns.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/
Gagolewski M., stringi: Fast and portable character string processing in R, Journal of Statistical
Software 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02
Other search_locate: about_search, stri_locate_all_boundaries()
Other indexing: stri_locate_all_boundaries(), stri_sub_all(), stri_sub()
Examples

```r
stri_locate_all('stringi', fixed='i')

stri_locate_first_coll('hladn\u00FD', 'HLADNY', strength=1, locale='sk_SK')

stri_locate_all_regex(c('breakfast=eggs;lunch=pizza', 'breakfast=spam', 'no food here'),
                      '(?<when>\w+)=(?<what>\w+)',
                      capture_groups=TRUE
                      ) # named capture groups

stri_locate_all_fixed("abababa", "ABA", case_insensitive=TRUE, overlap=TRUE)
stri_locate_first_fixed("ababa", "aba")
stri_locate_last_fixed("ababa", "aba") # starts from end
stri_locate_last_regex("ababa", "aba") # no overlaps, from left to right

x <- c("yes yes", "no", NA)
stri_locate_all_fixed(x, "yes")
stri_locate_all_fixed(x, "yes", omit_no_match=TRUE)
stri_locate_all_fixed(x, "yes", get_length=TRUE)
stri_locate_all_fixed(x, "yes", get_length=TRUE, omit_no_match=TRUE)
stri_locate_first_fixed(x, "yes")
stri_locate_first_fixed(x, "yes", get_length=TRUE)

# Use regex positive-lookahead to locate overlapping pattern matches:
stri_locate_all_regex("ACAGAGACCTTATAGAGAGA", '(?=AGA)')
# note that start > end here (match of length zero)
```

---

**stri_locate_all_boundaries**

*Locate Text Boundaries*

**Description**

These functions locate text boundaries (like character, word, line, or sentence boundaries). Use `stri_locate_all_*` to locate all the matches. `stri_locate_first_*` and `stri_locate_last_*` give the first or the last matches, respectively.

**Usage**

```r
stri_locate_all_boundaries(
  str,
  omit_no_match = FALSE,
  get_length = FALSE,
  ...,
  opts_brkiter = NULL
)
```
stri_locate_last_boundaries(str, get_length = FALSE, ..., opts_brkiter = NULL)
stri_locate_first_boundaries(str, get_length = FALSE, ..., opts_brkiter = NULL)
stri_locate_all_words(
  str,
  omit_no_match = FALSE,
  locale = NULL,
  get_length = FALSE
)
stri_locate_last_words(str, locale = NULL, get_length = FALSE)
stri_locate_first_words(str, locale = NULL, get_length = FALSE)

Arguments

- **str**: character vector or an object coercible to
- **omit_no_match**: single logical value; if TRUE, a no-match will be indicated by a matrix with 0 rows stri_locate_all_* only
- **get_length**: single logical value; if FALSE (default), generate from-to matrices; otherwise, output from-length ones
- **...**: additional settings for opts_brkiter
- **opts_brkiter**: named list with ICU BreakIterator's settings, see stri_opts_brkiter; NULL for default break iterator, i.e., line_break
- **locale**: NULL or ' ' for text boundary analysis following the conventions of the default locale, or a single string with locale identifier, see stringi-locale

Details

Vectorized over str.

For more information on text boundary analysis performed by ICU’s BreakIterator, see stringi-search-boundaries.

For stri_locate_*_words, just like in stri_extract_all_words and stri_count_words, ICU’s word BreakIterator iterator is used to locate the word boundaries, and all non-word characters (UBRK_WORD_NONE rule status) are ignored. This function is equivalent to a call to stri_locate_*_boundaries(str, type='word', skip_word_none=TRUE, locale=locale)

Value

stri_locate_all_* yields a list of length(str) integer matrices. stri_locate_first_* and stri_locate_last_* generate return an integer matrix. See stri_locate for more details.

Author(s)

Marek Gagolewski and other contributors
See Also

The official online manual of \texttt{stringi} at \url{https://stringi.gagolewski.com/}

Gagolewski M., \texttt{stringi}: Fast and portable character string processing in R, \textit{Journal of Statistical Software} 103(2), 2022, 1-59, \url{doi:10.18637/jss.v103.i02}

Other search_locate: \texttt{about_search}, \texttt{stri_locate_all()}

Other indexing: \texttt{stri_locate_all()}, \texttt{stri_sub_all()}, \texttt{stri_sub()}

Other locale_sensitive: \texttt{%s%<%()}\texttt{,about_locale,about_search_boundaries,about_search_coll,stri_compare(),stri_count_boundaries(),stri_duplicated(),stri_enc_detect2(),stri_extract_all_boundaries(),stri_opts_collator(),stri_order(),stri_rank(),stri_sort_key(),stri_sort(),stri_split_boundaries(),stri_trans_tolower(),stri_unique(),stri_wrap()}

Other text_boundaries: \texttt{about_search_boundaries,about_search,stri_count_boundaries(),stri_extract_all_boundaries(),stri_opts_brkiter(),stri_split_boundaries(),stri_split_lines(),stri_trans_tolower(),stri_wrap()}

Examples

```r
test <- 'The\u00a0above-mentioned features are very useful. Spam, spam, eggs, bacon, and spam.'
stri_locate_all_words(test)
stri_locate_all_boundaries(
  'Mr. Jones and Mrs. Brown are very happy. So am I, Prof. Smith.',
  type='sentence',
  locale='en_US@ss=standard' \# ICU >= 56 only
)
```

---

\texttt{stri_match_all} \hspace{1cm} \textit{Extract Regex Pattern Matches, Together with Capture Groups}

\underline{Description}

These functions extract substrings in \texttt{str} that match a given regex pattern. Additionally, they extract matches to every \texttt{capture group}, i.e., to all the sub-patterns given in round parentheses.

\underline{Usage}

\begin{verbatim}
stri_match_all(str, ...[, regex])
stri_match_first(str, ...[, regex])
stri_match_last(str, ...[, regex])
stri_match(str, ...[, regex, mode = c("first", "all", "last")])
stri_match_all_regex(
```
Arguments

- **str** character vector; strings to search in
- **pattern** character vector; search patterns; for more details refer to stringi-search
- **omit_no_match** single logical value; if FALSE, then a row with missing values will indicate that there was no match; stri_match_all_* only
- **cg_missing** single string to be used if a capture group match is unavailable
- **opts_regex** a named list with ICU Regex settings, see stri_opts_regex; NULL for default settings

Details

Vectorized over str and pattern (with recycling of the elements in the shorter vector if necessary). This allows to, for instance, search for one pattern in each given string, search for each pattern in one given string, and search for the i-th pattern within the i-th string.

If no pattern match is detected and omit_no_match=FALSE, then NAs are included in the resulting matrix (matrices), see Examples.

stri_match, stri_match_all, stri_match_first, and stri_match_last are convenience functions. They merely call stri_match_*_regex and are provided for consistency with other string searching functions’ wrappers, see, among others, stri_extract.
Value

For `stri_match_all*`, a list of character matrices is returned. Each list element represents the results of a different search scenario.

For `stri_match_first*` and `stri_match_last*` a character matrix is returned. Each row corresponds to a different search result.

The first matrix column gives the whole match. The second one corresponds to the first capture group, the third – the second capture group, and so on.

If regular expressions feature a named capture group, the matrix columns will be named accordingly. However, for `stri_match_first*` and `stri_match_last*` this will only be the case if there is a single pattern.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other search_extract: `about_search, stri_extract_all_boundaries(), stri_extract_all()`

Examples

`stri_match_all_regex('breakfast=eggs, lunch=pizza, dessert=icecream', '([^\w]=([^\w]+))')`
`stri_match_all_regex(c('breakfast=eggs', 'lunch=pizza', 'no food here'), '([^\w]=([^\w]+))')`
`stri_match_all_regex(c('breakfast=eggs;lunch=pizza', 'breakfast=bacon;lunch=spaghetti', 'no food here'), '([^\w]=([^\w]+))')`

`stri_match_first_regex(c('breakfast=eggs;lunch=pizza', 'breakfast=bacon;lunch=spaghetti', 'no food here'), '([^\w]=([^\w]+))', cg_missing='')`
`stri_match_last_regex(c('breakfast=eggs;lunch=pizza', 'breakfast=bacon;lunch=spaghetti', 'no food here'), '([^\w]=([^\w]+))')`

`stri_match_first_regex(c('abcd', ':abcd', ':abcd:'), '^:(?:[^:]*)?:$')`
`stri_match_first_regex(c('abcd', ':abcd', ':abcd:'), '^:(?:[^:]*)?:$', cg_missing='')`

# Match all the pattern of the form XYX, including overlapping matches:
`stri_match_all_regex('ACAGAGACTTTAGATAGAGAAGA', '([^ACGT][ACGT]\2)')[[1]][,2]`

# Compare the above to:
`stri_extract_all_regex('ACAGAGACTTTAGATAGAGAAGA', '([ACGT][ACGT]\1)')`
`stri_na2empty`  
*Replace NAs with Empty Strings*

**Description**

This function replaces all missing values with empty strings. See `stri_replace_na` for a generalization.

**Usage**

`stri_na2empty(x)`

**Arguments**

- `x`  
a character vector

**Value**

Returns a character vector.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other utils: `stri_list2matrix()`, `stri_remove_empty()`, `stri_replace_na()`

**Examples**

```r
stri_na2empty(c('a', NA, '', 'b'))
```
**stri_numbytes**  
*Count the Number of Bytes*

**Description**
Counts the number of bytes needed to store each string in the computer’s memory.

**Usage**

```r
stri_numbytes(str)
```

**Arguments**

- **str**: character vector or an object coercible to

**Details**
Often, this is not the function you would normally use in your string processing activities. See `stri_length` instead.

For 8-bit encoded strings, this is the same as `stri_length`. For UTF-8 strings, the returned values may be greater than the number of code points, as UTF-8 is not a fixed-byte encoding: one code point may be encoded by 1-4 bytes (according to the current Unicode standard).

Missing values are handled properly.

The strings do not need to be re-encoded to perform this operation.

The returned values do not include the trailing NUL bytes, which are used internally to mark the end of string data (in C).

**Value**

Returns an integer vector of the same length as `str`.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of stringi at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other length: `%s%()`, `stri_isempty()`, `stri_length()`, `stri_pad_both()`, `stri_sprintf()`, `stri_width()`
Examples

\begin{verbatim}
stri_numbytes(letters)
stri_numbytes(c('abc', '123', '\u0105\u0104'))

## Not run:
# this used to fail on Windows, where there were no native support
# for 4-bytes Unicode characters; see, however, stri_unescape_unicode():
stri_numbytes('\U001F600') # compare stri_length('\U001F600')

## End(Not run)
\end{verbatim}

---

**stri_opts_brkiter**  
*Generate a List with BreakIterator Settings*

**Description**

A convenience function to tune the ICU BreakIterator’s behavior in some text boundary analysis functions, see `stringi-search-boundaries`.

**Usage**

\begin{verbatim}
stri_opts_brkiter(
  type,
  locale,
  skip_word_none,
  skip_word_number,
  skip_word_letter,
  skip_word_kana,
  skip_word_ideo,
  skip_line_soft,
  skip_line_hard,
  skip_sentence_term,
  skip_sentence_sep
)
\end{verbatim}

**Arguments**

- **type** single string: either the break iterator type, one of character, line_break, sentence, word, or a custom set of ICU break iteration rules; see `stringi-search-boundaries`
- **locale** single string, NULL or '' for default locale
- **skip_word_none** logical; perform no action for ‘words’ that do not fit into any other categories
- **skip_word_number** logical; perform no action for words that appear to be numbers
skip_word_letter
  logical; perform no action for words that contain letters, excluding hiragana,
katakana, or ideographic characters

skip_word_kana
  logical; perform no action for words containing kana characters

skip_word_ideo
  logical; perform no action for words containing ideographic characters

skip_line_soft
  logical; perform no action for soft line breaks, i.e., positions where a line break
  is acceptable but not required

skip_line_hard
  logical; perform no action for hard, or mandatory line breaks

skip_sentence_term
  logical; perform no action for sentences ending with a sentence terminator (',',
  ';','?','!'), possibly followed by a hard separator (CR, LF, PS, etc.)

skip_sentence_sep
  logical; perform no action for sentences that do not contain an ending sentence
  terminator, but are ended by a hard separator or end of input

Details

The skip_* family of settings may be used to prevent performing any special actions on particular
styles of text boundaries, e.g., in case of the stri_locate_all_boundaries and stri_split_boundaries
functions.

Note that custom break iterator rules (advanced users only) should be specified as a single string.
For a detailed description of the syntax of RBBI rules, please refer to the ICU User Guide on
Boundary Analysis.

Value

Returns a named list object. Omitted skip_* values act as they have been set to FALSE.

Author(s)

Marek Gagolewski and other contributors

References

apidoc/dev/icu4c/ubrk_8h.html


See Also

The official online manual of stringi at https://stringi.gagolewski.com/

Gagolewski M., stringi: Fast and portable character string processing in R, Journal of Statistical
Software 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other text_boundaries: about_search_boundaries, about_search, stri_count_boundaries(),
stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_split_boundaries(),
stri_split_lines(), stri_trans_tolower(), stri_wrap()
Generate a List with Collator Settings

Description

A convenience function to tune the ICU Collator’s behavior, e.g., in `stri_compare`, `stri_order`, `stri_unique`, `stri_duplicated`, as well as `stri_detect_coll` and other `stringi-search-coll` functions.

Usage

```r
stri_opts_collator(
  locale = NULL,
  strength = 3L,
  alternate_shifted = FALSE,
  french = FALSE,
  uppercase_first = NA,
  case_level = FALSE,
  normalization = FALSE,
  normalisation = normalization,
  numeric = FALSE
)
```

```r
stri_coll(
  locale = NULL,
  strength = 3L,
  alternate_shifted = FALSE,
  french = FALSE,
  uppercase_first = NA,
  case_level = FALSE,
  normalization = FALSE,
  normalisation = normalization,
  numeric = FALSE
)
```

Arguments

- `locale` single string, `NULL` or `'` for default locale
- `strength` single integer in `{1,2,3,4}`, which defines collation strength; 1 for the most permissive collation rules, 4 for the strictest ones
- `alternate_shifted` single logical value; FALSE treats all the code points with non-ignorable primary weights in the same way, TRUE causes code points with primary weights that are equal or below the variable top value to be ignored on primary level and moved to the quaternary level
- `french` single logical value; used in Canadian French; TRUE results in secondary weights being considered backwards
**stri_opts_collator**

- **uppercase_first**: single logical value; NA orders upper and lower case letters in accordance to their tertiary weights, TRUE forces upper case letters to sort before lower case letters, FALSE does the opposite.

- **case_level**: single logical value; controls whether an extra case level (positioned before the third level) is generated or not.

- **normalization**: single logical value; if TRUE, then incremental check is performed to see whether the input data is in the FCD form. If the data is not in the FCD form, incremental NFD normalization is performed.

- **normalisation**: alias of normalization.

- **numeric**: single logical value; when turned on, this attribute generates a collation key for the numeric value of substrings of digits; this is a way to get ‘100’ to sort AFTER ‘2’; note that negative or non-integer numbers will not be ordered properly.

**Details**

ICU’s **collator** performs a locale-aware, natural-language alike string comparison. This is a more reliable way of establishing relationships between strings than the one provided by base R, and definitely one that is more complex and appropriate than ordinary bytewise comparison.

**Value**

Returns a named list object; missing settings are left with default values.

**Author(s)**

Marek Gagolewski and other contributors.

**References**

- icu::Collator Class Reference – ICU4C API Documentation, [https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1Collator.html](https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1Collator.html)

**See Also**

The official online manual of **stringi** at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, [doi:10.18637/jss.v103.i02](https://doi.org/10.18637/jss.v103.i02)

Other locale_sensitive: %s<%(), about_locale, about_search_boundaries, about_search_coll, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_order(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()

Other search_coll: about_search_coll, about_search
Examples

```r
stri_cmp('number100', 'number2')
stri_cmp('number100', 'number2', opts_collator=stri_opts_collator(numeric=TRUE))
stri_cmp('number100', 'number2', numeric=TRUE) # equivalent
stri_cmp('above mentioned', 'above-mentioned')
stri_cmp('above mentioned', 'above-mentioned', alternate_shifted=TRUE)
```

stri_opts_fixed  Generate a List with Fixed Pattern Search Engine’s Settings

Description

A convenience function used to tune up the behavior of `stri_*_fixed` functions, see `stringi-search-fixed`.

Usage

```r
stri_opts_fixed(case_insensitive = FALSE, overlap = FALSE)
```

Arguments

- `case_insensitive`
  - logical; enable simple case insensitive matching
- `overlap`
  - logical; enable overlapping matches’ detection

Details

Case-insensitive matching uses a simple, single-code point case mapping (via ICU’s `u_toupper()` function). Full case mappings should be used whenever possible because they produce better results by working on whole strings. They also take into account the string context and the language, see `stringi-search-coll`.

Searching for overlapping pattern matches is available in `stri_extract_all_fixed`, `stri_locate_all_fixed`, and `stri_count_fixed` functions.

Value

Returns a named list object.

Author(s)

Marek Gagolewski and other contributors

References

**stri_opts_regex**

*Generate a List with Regex Matcher Settings*

**Description**

A convenience function to tune the **ICU** regular expressions matcher’s behavior, e.g., in `stri_count_regex` and other `stringi-search-regex` functions.

**Usage**

```r
stri_opts_regex(
  case_insensitive,
  comments,
  dotall,
  dot_all = dotall,
  literal,
  multiline,
  multi_line = multiline,
  unix_lines,
  uword,
  error_on_unknown_escapes,
  time_limit = 0L,
  stack_limit = 0L
)
```

**Arguments**

- `case_insensitive` logical; enables case insensitive matching [regex flag `(?i)`]
- `comments` logical; allows white space and comments within patterns [regex flag `(?x)`]
- `dotall` logical; if set, ‘.’ matches line terminators, otherwise matching of ‘.’ stops at a line end [regex flag `(?s)`]
- `dot_all` alias of `dotall`
literal logical; if set, treat the entire pattern as a literal string; metacharacters or escape sequences in the input sequence will be given no special meaning; note that in most cases you would rather use the `stringi-search-fixed` facilities in this case

multiline logical; controls the behavior of `$` and `. `. If set, recognize line terminators within a string, otherwise, match only at start and end of input string [regex flag `(?m)`]

multi_line alias of `multiline`

unix_lines logical; Unix-only line endings; when enabled, only U+000a is recognized as a line ending by `. `. `$`, and `. `. [regex flag `(?m)`]

uword logical; Unicode word boundaries; if set, uses the Unicode TR 29 definition of word boundaries; warning: Unicode word boundaries are quite different from traditional regex word boundaries. [regex flag `(?w)`] See https://unicode.org/reports/tr29/#Word_Boundaries

error_on_unknown_escapes logical; whether to generate an error on unrecognized backslash escapes; if set, fail with an error on patterns that contain backslash-escaped ASCII letters without a known special meaning; otherwise, these escaped letters represent themselves

time_limit integer; processing time limit, in ~milliseconds (but not precisely so, depends on the CPU speed), for match operations; setting a limit is desirable if poorly written regexes are expected on input; 0 for no limit

stack_limit integer; maximal size, in bytes, of the heap storage available for the match backtracking stack; setting a limit is desirable if poorly written regexes are expected on input; 0 for no limit

Details

Note that some regex settings may be changed using ICU regex flags inside regexes. For example, `'(?!pattern)` performs a case-insensitive match of a given pattern, see the ICU User Guide entry on Regular Expressions in the References section or `stringi-search-regex`.

Value

Returns a named list object; missing settings are left with default values.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_regex: about_search_regex, about_search

Examples

stri_detect_regex('ala', 'ALA') # case-sensitive by default
stri_detect_regex('ala', 'ALA', opts_regex=stri_opts_regex(case_insensitive=TRUE))
stri_detect_regex('ala', 'ALA', case_insensitive=TRUE) # equivalent
stri_detect_regex('ala', '(?i)ALA') # equivalent

stri_order

Ordering Permutation

Description

This function finds a permutation which rearranges the strings in a given character vector into the ascending or descending locale-dependent lexicographic order.

Usage

stri_order(str, decreasing = FALSE, na_last = TRUE, ..., opts_collator = NULL)

Arguments

str a character vector
decreasing a single logical value; should the sort order be nondecreasing (FALSE, default) or nonincreasing (TRUE)?
na_last a single logical value; controls the treatment of NAs in str. If TRUE, then missing values in str are put at the end; if FALSE, they are put at the beginning; if NA, then they are removed from the output
... additional settings for opts_collator
opts_collator a named list with ICU Collator’s options, see stri_opts_collator, NULL for default collation options

Details

For more information on ICU’s Collator and how to tune it up in stringi, refer to stri_opts_collator. As usual in stringi, non-character inputs are coerced to strings, see an example below for a somewhat non-intuitive behavior of lexicographic sorting on numeric inputs.

This function uses a stable sort algorithm (STL’s stable_sort), which performs up to \(N \cdot \log^2(N)\) element comparisons, where \(N\) is the length of str.

For ordering with regards to multiple criteria (such as sorting data frames by more than 1 column), see stri_rank.
**Value**

The function yields an integer vector that gives the sort order.

**Author(s)**

Marek Gagolewski and other contributors

**References**


**See Also**

The official online manual of *stringi* at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other locale_sensitive: %s<%(), about_locale, about_search_boundaries, about_search_coll, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_collator(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()

**Examples**

```r
stri_order(c('hladny', 'chladny'), locale='pl_PL')
stri_order(c('hladny', 'chladny'), locale='sk_SK')

stri_order(c(1, 100, 2, 101, 11, 10))  # lexicographic order
stri_order(c(1, 100, 2, 101, 11, 10), numeric=TRUE)  # OK for integers
stri_order(c(0.25, 0.5, 1, -1, -2, -3), numeric=TRUE)  # incorrect
```

---

**stri_pad_both**

*Pad (Center/Left/Right Align) a String*

**Description**

Add multiple pad characters at the given side(s) of each string so that each output string is of total width of at least width. These functions may be used to center or left/right-align each string.

**Usage**

```r
stri_pad_both(
  str,
  width = floor(0.9 * getOption("width")),
  pad = " ",
  use_length = FALSE
)
```
str_pad_both

str_pad_left(
  str,
  width = floor(0.9 *getOption("width")),
  pad = " ",
  use_length = FALSE
)

str_pad_right(
  str,
  width = floor(0.9 *getOption("width")),
  pad = " ",
  use_length = FALSE
)

str_pad(
  str,
  width = floor(0.9 *getOption("width")),
  side = c("left", "right", "both"),
  pad = " ",
  use_length = FALSE
)

Arguments

str character vector
width integer vector giving minimal output string lengths
pad character vector giving padding code points
use_length single logical value; should the number of code points be used instead of the total code point width (see stri_width)?
side [stri_pad only] single character string; sides on which padding character is added (left (default), right, or both)

Details

Vectorized over str, width, and pad. Each string in pad should consist of a code points of total width equal to 1 or, if use_length is TRUE, exactly one code point.

str_pad is a convenience function, which dispatches to str_pad_*.

Note that Unicode code points may have various widths when printed on the console and that, by default, the function takes that into account. By changing the state of the use_length argument, this function starts acting like each code point was of width 1. This feature should rather be used with text in Latin script.

See stri_trim_left (among others) for reverse operation. Also check out stri_wrap for line wrapping.

Value

These functions return a character vector.
**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other length: `%s$%()`, `stri_isempty()`, `stri_length()`, `stri_numbytes()`, `stri_sprintf()`, `stri_width()`

**Examples**

```r
stri_pad_left('stringi', 10, pad='#')
stri_pad_both('stringi', 8:12, pad='*')
# center on screen:
cat(stri_pad_both(c('the', 'string', 'processing', 'package'),
getOption('width')*0.9), sep='\n')
cat(stri_pad_both(c('한글', # takes width into account
stri_trans_nfkd('한글'), 'abcd'),
width=10), sep='\n')
```

---

**stri_rand_lipsum**

*A Lorem Ipsum Generator*

**Description**

Generates (pseudo)random *lorem ipsum* text consisting of a given number of text paragraphs.

**Usage**

```r
stri_rand_lipsum(n_paragraphs, start_lipsum = TRUE, nparagraphs = n_paragraphs)
```

**Arguments**

- `n_paragraphs` single integer, number of paragraphs to generate
- `start_lipsum` single logical value; should the resulting text start with *Lorem ipsum dolor sit amet?*
- `nparagraphs` [DEPRECATED] alias of `n_paragraphs`
stri_rand_shuffle

Details

Lorem ipsum is a dummy text often used as a source of data for string processing and displaying/layouting exercises.

The current implementation is very simple: words are selected randomly from a Zipf distribution (based on a set of ca. 190 predefined Latin words). The number of words per sentence and sentences per paragraph follows a discretized, truncated normal distribution. No Markov chain modeling, just i.i.d. word selection.

Value

Returns a character vector of length n_paragraphs.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other random: stri_rand_shuffle(), stri_rand_strings()

Examples

cat(sapply(
    stri_wrap(stri_rand_lipsum(10), 80, simplify=FALSE),
    stri_flatten, collapse='\n'), sep='\n
')
cat(stri_rand_lipsum(10), sep='\n
')

stri_rand_shuffle

Randomly Shuffle Code Points in Each String

Description

Generates a (pseudo)random permutation of the code points in each string.

Usage

stri_rand_shuffle(str)

Arguments

str character vector
stri_rand_strings

Generate Random Strings

Description

Generates (pseudo)random strings of desired lengths.

Usage

stri_rand_strings(n, length, pattern = "[A-Za-z0-9]")

Arguments

n

single integer, number of observations

length

integer vector, desired string lengths

pattern

character vector specifying character classes to draw elements from, see stringi-search-charclass
### stri_rank

**Description**

This function ranks each string in a character vector according to a locale-dependent lexicographic order. It is a portable replacement for the base `xtfrm` function.

**Details**

Vectorized over length and pattern. If length of length or pattern is greater than n, then redundant elements are ignored. Otherwise, these vectors are recycled if necessary.

This operation may result in non-Unicode-normalized strings and may give peculiar outputs for bidirectional strings.

Sampling of code points from the set specified by pattern is always done with replacement and each code point appears with equal probability.

**Value**

Returns a character vector.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other random: `stri_rand_lipsum()`, `stri_rand_shuffle()`

**Examples**

```r
stri_rand_strings(5, 10) # 5 strings of length 10
stri_rand_strings(5, sample(1:10, 5, replace=TRUE)) # 5 strings of random lengths
stri_rand_strings(10, 5, '\p{script=latin}\p{Ll}') # small letters from the Latin script

# generate n random passwords of length in [8, 14] # consisting of at least one digit, small and big ASCII letter:
n <- 10
stri_rand_shuffle(stri_paste(
  stri_rand_strings(n, 1, '[0-9]'),
  stri_rand_strings(n, 1, '[a-z]'),
  stri_rand_strings(n, 1, '[A-Z]'),
  stri_rand_strings(n, sample(5:11, 5, replace=TRUE), '[a-zA-Z0-9]')
))
```
Usage
stri_rank(str, ..., opts_collator = NULL)

Arguments
str  a character vector
... additional settings for opts_collator
opts_collator  a named list with ICU Collator's options, see stri_opts_collator, NULL for default collation options

Details
Missing values result in missing ranks and tied observations receive the same ranks (based on min).
For more information on ICU’s Collator and how to tune it up in stringi, refer to stri_opts_collator.

Value
The result is a vector of ranks corresponding to each string in str.

Author(s)
Marek Gagolewski and other contributors

References

See Also
The official online manual of stringi at https://stringi.gagolewski.com/

Other locale_sensitive: %s<%(), about_locale, about_search_boundaries, about_search_coll,
stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(),
stri_locate_all_boundaries(), stri_opts_collator(), stri_order(), stri_sort_key(),
stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_unique(), stri_wrap()

Examples
stri_rank(c('hladny', 'chladny'), locale='pl_PL')
stri_rank(c('hladny', 'chladny'), locale='sk_SK')

stri_rank("a" %s+% c(1, 100, 2, 101, 11, 10))  # lexicographic order
stri_rank("a" %s+% c(1, 100, 2, 101, 11, 10), numeric=TRUE)  # OK
stri_rank("a" %s+% c(0.25, 0.5, 1, -1, -2, -3), numeric=TRUE)  # incorrect

# Ordering a data frame with respect to two criteria:
X <- data.frame(a=c("b", NA, "b", "b", NA, "a", "a", "c"), b=runif(8))
X[order(stri_rank(X$a), X$b), ]
stri_read_lines

Read Text Lines from a Text File

Description

Reads a text file in its entirety, re-encodes it, and splits it into text lines.

Usage

stri_read_lines(con, encoding = NULL, fname = con)

Arguments

- con: name of the output file or a connection object (opened in the binary mode)
- encoding: single string; input encoding; NULL or '' for the current default encoding.
- fname: [DEPRECATED] alias of con

Details

This aims to be a substitute for the `readLines` function, with the ability to re-encode the input file in a much more robust way, and split the text into lines with `stri_split_lines1` (which conforms with the Unicode guidelines for newline markers).

The function calls `stri_read_raw`, `stri_encode`, and `stri_split_lines1`, in this order.

Because of the way this function is currently implemented, maximal file size cannot exceed ~0.67 GB.

Value

Returns a character vector, each text line is a separate string. The output is always marked as UTF-8.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other files: `stri_read_raw()`, `stri_write_lines()`
Description

Reads a text file as-is, with no conversion or text line splitting.

Usage

```r
stri_read_raw(con, fname = con)
```

Arguments

- `con` name of the output file or a connection object (opened in the binary mode)
- `fname` [DEPRECATED] alias of `con`

Details

Once a text file is read into memory, encoding detection (see `stri_enc_detect`), conversion (see `stri_encode`), and/or splitting of text into lines (see `stri_split_lines1`) can be performed.

Value

Returns a vector of type `raw`.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other files: `stri_read_lines()`, `stri_write_lines()`
**stri_remove_empty**  
*Remove All Empty Strings from a Character Vector*

**Description**

`stri_remove_empty` (alias `stri_omit_empty`) removes all empty strings from a character vector, and, if `na_empty` is TRUE, also gets rid of all missing values.

`stri_remove_empty_na` (alias `stri_omit_empty_na`) removes both empty strings and missing values.

`stri_remove_na` (alias `stri_omit_na`) returns a version of `x` with missing values removed.

**Usage**

```r
stri_remove_empty(x, na_empty = FALSE)
stri_omit_empty(x, na_empty = FALSE)
stri_remove_empty_na(x)
stri_omit_empty_na(x)
stri_remove_na(x)
stri_omit_na(x)
```

**Arguments**

- `x` : a character vector
- `na_empty` : should missing values be treated as empty strings?

**Value**

Returns a character vector.

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other utils: `stri_list2matrix()`, `stri_na2empty()`, `stri_replace_na()`
**Examples**

```r
c(r('a', NA, '', 'b'))
c(r('a', NA, '', 'b'))
c(r('a', NA, '', 'b'), TRUE)
```

```r
c(r('a', NA, '', 'b'))
c(r('a', NA, '', 'b'), TRUE)
```

```r
c(r('a', NA, '', 'b'))
```

**Description**

These functions replace, with the given replacement string, every/first/last substring of the input that matches the specified pattern.

**Usage**

```r
stri_replace_all(str, replacement, ..., regex, fixed, coll, charclass)
stri_replace_first(str, replacement, ..., regex, fixed, coll, charclass)
stri_replace_last(str, replacement, ..., regex, fixed, coll, charclass)
stri_replace(str, replacement, ..., regex, fixed, coll, charclass, mode = c("first", "all", "last")

stri_replace_all_charclass(str, pattern, replacement)
stri_replace_first_charclass(str, pattern, replacement)
stri_replace_last_charclass(str, pattern, replacement)
```
Arguments

str: character vector; strings to search in
replacement: character vector with replacements for matched patterns
...: supplementary arguments passed to the underlying functions, including addi-
stri_replace_all

tional settings for opts_collator, opts_regex, opts_fixed, and so on

mode

single string; one of: 'first' (the default), 'all', 'last'

pattern, regex, fixed, coll, charclass

character vector; search patterns; for more details refer to stringi-search

merge

single logical value; should consecutive matches be merged into one string;
stri_replace_all_charclass only

vectorize_all

single logical value; should each occurrence of a pattern in every string be re-
placed by a corresponding replacement string?; stri_replace_all_* only

vectorise_all

alias of vectorize_all

opts_collator, opts_fixed, opts_regex

a named list used to tune up the search engine’s settings; see stri_opts_collator,
stri_opts_fixed, and stri_opts_regex, respectively; NULL for the defaults

Details

By default, all the functions are vectorized over str, pattern, replacement (with recycling of the elements in the shorter vector if necessary). Input that is not part of any match is left unchanged; each match is replaced in the result by the replacement string.

However, for stri_replace_all*, if vectorize_all is FALSE, then each substring matching any of the supplied patterns is replaced by a corresponding replacement string. In such a case, the vectorization is over str, and - independently - over pattern and replacement. In other words, this is equivalent to something like for (i in 1:npatterns) str <- stri_replace_all(str, pattern[i], replacement[i]). Note that you must set length(pattern) >= length(replacement).

In case of stri_replace_*.regex, the replacement string may contain references to capture groups (in round parentheses). References are of the form $n, where n is the number of the capture group ($1 denotes the first group). For the literal $, escape it with a backslash. Moreover, ${name} are used for named capture groups.

Note that stri_replace_last_regex searches from start to end, but skips overlapping matches, see the example below.

str_replace, stri_replace_all, stri_replace_first, and stri_replace_last are conve-
nience functions; they just call stri_replace_*_* variants, depending on the arguments used.

If you wish to remove white-spaces from the start or end of a string, see stri_trim.

Value

All the functions return a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_replace: about_search, stri_replace_rstr(), stri_trim_both()
Examples

```r
stri_replace_all_charclass('aaaa', '[a]', 'b', merge=c(TRUE, FALSE))
stri_replace_all_charclass('a\n\n\tc \d', '\\p{WHITE_SPACE}', '\ ') 
stri_replace_all_charclass('a\n\n\tc \d', '\\p{WHITE_SPACE}', ' ', merge=TRUE)

s <- 'Lorem ipsum dolor sit amet, consectetur adipisicing elit.'
stri_replace_all_fixed(s, '', '#')
stri_replace_all_fixed(s, 'o', '0')

stri_replace_all_fixed(c('1', 'NULL', '3'), 'NULL', NA)
stri_replace_all_regex(s, '\s?\s', '#')
stri_replace_all_regex(s, '(el)\s?it', '1234')
stri_replace_all_regex('abaca', 'a', c('!', '*'))
stri_replace_all_regex('123[456]789', '(\p{N})\d+', '$2-$1')
stri_replace_all_regex(c('stringi R', 'REXAMINE', '123'), 'R|R.', 'r')

# named capture groups are available since ICU 55
## Not run:
## Compare the results:
stri_replace_all_fixed('The quick brown fox jumped over the lazy dog.',
  c('quick', 'brown', 'fox'), c('slow', 'black', 'bear'), vectorize_all=TRUE)
stri_replace_all_fixed('The quick brown fox jumped over the lazy dog.',
  c('quick', 'brown', 'fox'), c('slow', 'black', 'bear'), vectorize_all=FALSE)

# Compare the results:
stri_replace_all_fixed('The quicker brown fox jumped over the lazy dog.',
  c('quick', 'brown', 'fox'), c('slow', 'black', 'bear'), vectorize_all=FALSE)
stri_replace_all_regex('The quicker brown fox jumped over the lazy dog.',
  '\b\s+\w+\w+\b', c('quick', 'brown', 'fox')%s%+\b, c('slow', 'black', 'bear'), vectorize_all=FALSE)

# Searching for the last occurrence:
# Note the difference - regex searches left to right, with no overlaps.
stri_replace_last_fixed("agAGA", "aga", "*", case_insensitive=TRUE)
stri_replace_last_regex("agAGA", "aga", "*", case_insensitive=TRUE)
```

### Description

This function gives a convenient way to replace each missing (NA) value with a given string.
Usage

stri_replace_na(str, replacement = "NA")

Arguments

- **str** character vector or an object coercible to
- **replacement** single string

Details

This function is roughly equivalent to

```
str2 <- stri_enc_toutf8(str); str2[is.na(str2)] <- stri_enc_toutf8(replacement); str2
```

It may be used, e.g., wherever the 'plain R' NA handling is desired, see Examples.

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of **stringi** at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, [doi:10.18637/jss.v103.i02](https://doi.org/10.18637/jss.v103.i02)

Other utils: `stri_list2matrix()`, `stri_na2empty()`, `stri_remove_empty()`

Examples

```r
x <- c('test', NA)
stri_paste(x, 1:2) # 'test' NA
paste(x, 1:2) # 'test 1' 'NA 2'
stri_paste(stri_replace_na(x), 1:2, sep=' ') # 'test 1' 'NA 2'
```

---

**stri_replace_rstr** Convert gsub-Style Replacement Strings

Description

Converts a `gsub`-style replacement strings to those which can be used in `stri_replace`. In particular, `$` becomes `\$` and `\` becomes `$1`.

Usage

```
stri_replace_rstr(x)
```
Arguments

x character vector

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/
Other search_replace: about_search, stri_replace_all(), stri_trim_both()

Description

Reverses the order of the code points in every string.

Usage

stri_reverse(str)

Arguments

str character vector

Details

Note that this operation may result in non-Unicode-normalized strings and may give peculiar outputs for bidirectional strings.
See also stri_rand_shuffle for a random permutation of code points.

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors
See Also
The official online manual of stringi at https://stringi.gagolewski.com/

Examples
stri_reverse(c('123', 'abc d e f'))
stri_reverse('ZXY (\u0105\u0104123$^\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·
**stri_sort_key**

**Description**

This function computes a locale-dependent sort key, which is an alternative character representation of the string that, when ordered in the C locale (which orders using the underlying bytes directly), will give an equivalent ordering to the original string. It is useful for enhancing algorithms that sort only in the C locale (e.g., the `strcmp` function in libc) with the ability to be locale-aware.

**Usage**

```r
stri_sort_key(str, ..., opts_collator = NULL)
```

**Arguments**

- `str` a character vector
- `...` additional settings for `opts_collator`
- `opts_collator` a named list with **ICU** Collator’s options, see `stri_opts_collator`, NULL for default collation options

**Examples**

```r
stri_sort(c("hladny", "chladny"), locale="pl_PL")
stri_sort(c("hladny", "chladny"), locale="sk_SK")
stri_sort(sample(LETTERS))
stri_sort(c(1, 100, 2, 101, 11, 10)) # lexicographic order
stri_sort(c(1, 100, 2, 101, 11, 10), numeric=TRUE) # OK for integers
stri_sort(c(0.25, 0.5, 1, -1, -2, -3), numeric=TRUE) # incorrect
```
Details

For more information on ICU’s Collator and how to tune it up in stringi, refer to stri_opts_collator.
See also stri_rank for ranking strings with a single character vector, i.e., generating relative sort keys.

Value

The result is a character vector with the same length as str that contains the sort keys. The output is marked as bytes-encoded.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other locale_sensitive: %s<%(), about_locale, about_search_boundaries, about_search_coll, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_sort(), stri_sort_key()

Examples

stri_sort_key(c('hladny', 'chladny'), locale='pl_PL')
stri_sort_key(c('hladny', 'chladny'), locale='sk_SK')

Description

These functions split each element in str into substrings. pattern defines the delimiters that separate the inputs into tokens. The input data between the matches become the fields themselves.
Usage

\texttt{stri_split(str, ..., regex, fixed, coll, charclass)}

\texttt{stri_split_fixed(}
  \texttt{str,}
  \texttt{pattern,}
  \texttt{n = -1L,}
  \texttt{omit_empty = FALSE,}
  \texttt{tokens_only = FALSE,}
  \texttt{simplify = FALSE,}
  \texttt{...}
  \texttt{opts_fixed = NULL}
\texttt{)}

\texttt{stri_split_regex(}
  \texttt{str,}
  \texttt{pattern,}
  \texttt{n = -1L,}
  \texttt{omit_empty = FALSE,}
  \texttt{tokens_only = FALSE,}
  \texttt{simplify = FALSE,}
  \texttt{...}
  \texttt{opts_regex = NULL}
\texttt{)}

\texttt{stri_split_coll(}
  \texttt{str,}
  \texttt{pattern,}
  \texttt{n = -1L,}
  \texttt{omit_empty = FALSE,}
  \texttt{tokens_only = FALSE,}
  \texttt{simplify = FALSE,}
  \texttt{...}
  \texttt{opts_collator = NULL}
\texttt{)}

\texttt{stri_split_charclass(}
  \texttt{str,}
  \texttt{pattern,}
  \texttt{n = -1L,}
  \texttt{omit_empty = FALSE,}
  \texttt{tokens_only = FALSE,}
  \texttt{simplify = FALSE}
\texttt{)}

Arguments

\texttt{str} character vector; strings to search in
stri_split

... supplementary arguments passed to the underlying functions, including additional settings for opts_collator, opts_regex, opts_fixed, and so on

pattern, regex, fixed, coll, charclass
character vector; search patterns; for more details refer to stringi-search

n integer vector, maximal number of strings to return, and, at the same time, maximal number of text boundaries to look for

omit_empty logical vector; determines whether empty tokens should be removed from the result (TRUE or FALSE) or replaced with NAs (NA)

tokens_only single logical value; may affect the result if n is positive, see Details

simplify single logical value; if TRUE or NA, then a character matrix is returned; otherwise (the default), a list of character vectors is given, see Value

opts_collator, opts_fixed, opts_regex
a named list used to tune up the search engine’s settings; see stri_opts_collator, stri_opts_fixed, and stri_opts_regex, respectively; NULL for the defaults

Details

Vectorized over str, pattern, n, and omit_empty (with recycling of the elements in the shorter vector if necessary).

If n is negative, then all pieces are extracted. Otherwise, if tokens_only is FALSE (which is the default), then n-1 tokens are extracted (if possible) and the n-th string gives the remainder (see Examples). On the other hand, if tokens_only is TRUE, then only full tokens (up to n pieces) are extracted.

omit_empty is applied during the split process: if it is set to TRUE, then tokens of zero length are ignored. Thus, empty strings will never appear in the resulting vector. On the other hand, if omit_empty is NA, then empty tokens are substituted with missing strings.

Empty search patterns are not supported. If you wish to split a string into individual characters, use, e.g., stri_split_boundaries(str, type='character') for THE Unicode way.

stri_split is a convenience function. It calls either stri_split_regex, stri_split_fixed, stri_split_coll, or stri_split_charclass, depending on the argument used.

Value

If simplify=FALSE (the default), then the functions return a list of character vectors.

Otherwise, stri_list2matrix with byrow=TRUE and n_min=n arguments is called on the resulting object. In such a case, a character matrix with an appropriate number of rows (according to the length of str, pattern, etc.) is returned. Note that stri_list2matrix’s fill argument is set to an empty string and NA, for simplify equal to TRUE and NA, respectively.

Author(s)

Marek Gagolewski and other contributors
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_split: about_search, stri_split_boundaries(), stri_split_lines()

Examples

```r
stri_split_fixed('a_b_c_d', '\_')
stri_split_fixed('a_b_c__d', '\_')
stri_split_fixed('a_b_c__d', '\_', omit_empty=TRUE)
stri_split_fixed('a_b_c__d', '\_', n=2, tokens_only=FALSE) # 'a' & remainder
stri_split_fixed('a_b_c__d', '\_', n=2, tokens_only=TRUE) # 'a' & 'b' only
stri_split_fixed('a_b_c__d', '\_', n=4, omit_empty=TRUE, tokens_only=TRUE)
stri_split_fixed('a_b_c__d', '\_', n=4, omit_empty=FALSE, tokens_only=TRUE)
stri_split_fixed('a_b_c__d', '\_', omit_empty=NA)
stri_split_fixed(c('ab,c', 'd,ef,g', 'h', ''), '\_', n=1, tokens_only=TRUE, omit_empty=TRUE)
stri_split_fixed(c('ab,c', 'd,ef,g', 'h', ''), '\_', n=2, tokens_only=TRUE, omit_empty=TRUE)
stri_split_fixed(c('ab,c', 'd,ef,g', 'h', ''), '\_', n=3, tokens_only=TRUE, omit_empty=TRUE)

stri_list2matrix(stri_split_fixed(c('ab,c', 'd,ef,g', '\h', ''), '\', omit_empty=TRUE))
stri_split_fixed(c('ab,c', 'd,ef,g', '\h', ''), '\', omit_empty=FALSE, simplify=TRUE)
stri_split_fixed(c('ab,c', 'd,ef,g', '\h', ''), '\', omit_empty=NA, simplify=TRUE)
stri_split_fixed(c('ab,c', 'd,ef,g', '\h', ''), '\', omit_empty=TRUE, simplify=TRUE)

stri_split_regex(c('ab,c', 'd,ef,g', '\h', ''),
                '\p{WHITE_SPACE}*','\p{WHITE_SPACE}*', omit_empty=NA, simplify=TRUE)
stri_split_charclass('Lorem ipsum dolor sit amet','\p{WHITE_SPACE}')
stri_split_charclass('Lorem ipsum dolor ','\p{WHITE_SPACE}', n=3,
                    omit_empty=c(FALSE, TRUE))
stri_split_regex('Lorem ipsum dolor sit amet','\p{Z}+') # see also stri_split_charclass
```

Description

This function locates text boundaries (like character, word, line, or sentence boundaries) and splits strings at the indicated positions.
Usage

```r
stri_split_boundaries(
  str,
  n = -1L,
  tokens_only = FALSE,
  simplify = FALSE,
  ...,
  opts_brkiter = NULL
)
```

Arguments

- `str`: character vector or an object coercible to
- `n`: integer vector, maximal number of strings to return
- `tokens_only`: single logical value; may affect the result if `n` is positive, see Details
- `simplify`: single logical value; if TRUE or NA, then a character matrix is returned; otherwise (the default), a list of character vectors is given, see Value
- `...`: additional settings for `opts_brkiter`
- `opts_brkiter`: a named list with ICU BreakIterator's settings, see `stri_opts_brkiter`; NULL for the default break iterator, i.e., `line_break`

Details

Vectorized over `str` and `n`.

If `n` is negative (the default), then all text pieces are extracted.

Otherwise, if `tokens_only` is FALSE (which is the default), then `n-1` tokens are extracted (if possible) and the `n`-th string gives the (non-split) remainder (see Examples). On the other hand, if `tokens_only` is TRUE, then only full tokens (up to `n` pieces) are extracted.

For more information on text boundary analysis performed by ICU's BreakIterator, see `stringi_search_boundaries`.

Value

If `simplify`=FALSE (the default), then the functions return a list of character vectors.

Otherwise, `stri_list2matrix` with `byrow=TRUE` and `n_min=n` arguments is called on the resulting object. In such a case, a character matrix with `length(str)` rows is returned. Note that `stri_list2matrix`'s `fill` argument is set to an empty string and NA, for `simplify` equal to TRUE and NA, respectively.

Author(s)

Marek Gagolewski and other contributors


**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other search_split: `about_search`, `stri_split_lines()`, `stri_split()`

Other locale_sensitive: `%s<%`, `about_locale`, `about_search_coll`, `stri_compare()`, `stri_count_boundaries()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_sort_key()`, `stri_sort()`, `stri_trans_tolower()`, `stri_unique()`, `stri_wrap()`

Other text_boundaries: `about_search_boundaries`, `about_search`, `stri_count_boundaries()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_sort_key()`, `stri_sort()`, `stri_wrap()`, `stri_split_lines()`, `stri_trans_tolower()`, `stri_unique()`, `stri_wrap()`

**Examples**

test <- 'The above-mentioned features are very useful. ' %s+% 'Spam, spam, eggs, bacon, and spam. 123 456 789'
stri_split_boundaries(test, type='line')
stri_split_boundaries(test, type='word')
stri_split_boundaries(test, type='word', skip_word_none=TRUE)
stri_split_boundaries(test, type='word', skip_word_none=TRUE, skip_word_letter=TRUE)
stri_split_boundaries(test, type='word', skip_word_none=TRUE, skip_word_number=TRUE)
stri_split_boundaries(test, type='sentence')
stri_split_boundaries(test, type='sentence', skip_sentence_sep=TRUE)
stri_split_boundaries(test, type='character')

# a filtered break iterator with the new ICU:
stri_split_boundaries('Mr. Jones and Mrs. Brown are very happy.
So am I, Prof. Smith.', type='sentence', locale='en_US@ss=standard') # ICU >= 56 only

---

**stri_split_lines**  
*Split a String Into Text Lines*

**Description**

These functions split each character string in a given vector into text lines.

**Usage**

```r
stri_split_lines(str, omit_empty = FALSE)
stri_split_lines1(str)
```
Arguments

str  character vector (stri_split_lines) or a single string (stri_split_lines1)
omit_empty  logical vector; determines whether empty strings should be removed from the result [stri_split_lines only]

Details

Vectorized over str and omit_empty.
omit_empty is applied when splitting. If set to TRUE, then empty strings will never appear in the resulting vector.

Newlines are represented with the Carriage Return (CR, 0xD), Line Feed (LF, 0xA), CRLF, or Next Line (NEL, 0x85) characters, depending on the platform. Moreover, the Unicode Standard defines two unambiguous separator characters, the Paragraph Separator (PS, 0x2029) and the Line Separator (LS, 0x2028). Sometimes also the Vertical Tab (VT, 0xB) and the Form Feed (FF, 0xC) are used for this purpose.

These stringi functions follow UTR#18 rules, where a newline sequence corresponds to the following regular expression: (?:\u{D A}|(?!\u{D A})[\u{A}-\u{D}\u{85}\u{2028}\u{2029}]. Each match serves as a text line separator.

Value

stri_split_lines returns a list of character vectors. If any input string is NA, then the corresponding list element is a single NA string.
stri_split_lines1(str) is equivalent to stri_split_lines(str[1])[1] (with default parameters), therefore it returns a character vector. Moreover, if the input string ends with a newline sequence, the last empty string is omitted from the file’s contents into text lines.

Author(s)

Marek Gagolewski and other contributors

References

Unicode Regular Expressions – Unicode Technical Standard #18, https://www.unicode.org/reports/tr18/

See Also

The official online manual of stringi at https://stringi.gagolewski.com/
Other search_split: about_search, stri_split_boundaries(), stri_split()
Other text_boundaries: about_search_boundaries, about_search, stri_count_boundaries(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_brkiter(), stri_split_boundaries(), stri_trans_tolower(), stri_wrap()
**Description**

`stri_sprintf` (synonym: `stri_string_format`) is a Unicode-aware replacement for and enhancement of the built-in `sprintf` function. Moreover, `stri_printf` prints formatted strings.

**Usage**

```r
stri_sprintf(
  format,
  ..., 
  na_string = NA_character_,
  inf_string = "Inf",
  nan_string = "NaN",
  use_length = FALSE
)

stri_string_format(
  format,
  ..., 
  na_string = NA_character_,
  inf_string = "Inf",
  nan_string = "NaN",
  use_length = FALSE
)

stri_printf(
  format,
  ..., 
  file = "",
  sep = "\n",
  append = FALSE,
  na_string = "NA",
  inf_string = "Inf",
  nan_string = "NaN",
  use_length = FALSE
)
```

**Arguments**

- `format` character vector of format strings
- `...` vectors (coercible to integer, real, or character)
- `na_string` single string to represent missing values; if NA, missing values in ... result in the corresponding outputs be missing too; use "NA" for compatibility with base R
\texttt{inf\_string} single string to represent the (unsigned) infinity (\texttt{NA} allowed)
\texttt{nan\_string} single string to represent the not-a-number (\texttt{NA} allowed)
\texttt{use\_length} single logical value; should the number of code points be used when applying modifiers such as \texttt{%20s} instead of the total code point width?

\texttt{file} \hspace{1cm} \texttt{see cat}
\texttt{sep} \hspace{1cm} \texttt{see cat}
\texttt{append} \hspace{1cm} \texttt{see cat}

**Details**

Vectorized over \texttt{format} and all vectors passed via \ldots.

Unicode code points may have various widths when printed on the console (compare \texttt{stri\_width}). These functions, by default (see the \texttt{use\_length} argument), take this into account.

These functions are not locale sensitive. For instance, numbers are always formatted in the "POSIX" style, e.g., \texttt{-123456.789} (no thousands separator, dot as a fractional separator). Such a feature might be added at a later date, though.

All arguments passed via \ldots are evaluated. If some of them are unused, a warning is generated. Too few arguments result in an error.

Note that \texttt{stri\_printf} treats missing values in \ldots as "\texttt{NA}" strings by default.

All format specifiers supported \texttt{sprintf} are also available here. For the formatting of integers and floating-point values, currently the system \texttt{std\::snprintf()} is called, but this may change in the future. Format specifiers are normalized and necessary sanity checks are performed.

Supported conversion specifiers: \texttt{dixoX} (integers) \texttt{feEgGa} (floats) and \texttt{s} (character strings). Supported flags: \texttt{-} (left-align), \texttt{+} (force output sign or blank when \texttt{NaN} or \texttt{NA}; numeric only), \texttt{<space>} (output minus or space for a sign; numeric only) \texttt{0} (pad with 0s; numeric only), \texttt{#} (alternative output of some numerics).

**Value**

\texttt{stri\_printf} is used for its side effect, which is printing text on the standard output or other connection/file. Hence, it returns \texttt{invisible(NULL)}.

The other functions return a character vector.

**Author(s)**

Marek Gagolewski and other contributors

**References**

\texttt{printf} in glibc, \url{https://man.archlinux.org/man/printf.3}
\texttt{printf} format strings – Wikipedia, \url{https://en.wikipedia.org/wiki/Printf_format_string}
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other length: %ss(), stri_isempty(), stri_length(), stri_numbytes(), stri_pad_both(), stri_width()

Examples

```r
stri_printf("%4s=%.3f", c("e", "e\u00b2", "\u03c0", "\u03c0\u00b2"),
  c(exp(1), exp(2), pi, pi^2))
```

```r
x <- c(
  "xxabcd",
  "xx\u0105\u0106\u0107\u0108",
  stri_paste(
    "\u200b\u200b\u200b\u200b",
    "\U0001F3F4\U000E0067\U000E0062\U000E0073\U000E0063\U000E0074\U000E007F",
    "abcd"
  ))
stri_printf("[%10s]", x) # minimum width = 10
stri_printf("[%-10.3s]", x) # output of max width = 3, but pad to width of 10
stri_printf("[%s]", x, use_length=TRUE) # minimum number of Unicode code points = 10

# vectorization wrt all arguments:
p <- runif(10)
stri_sprintf(ifelse(p > 0.5, "P(Y=1)=%1$.2f", "P(Y=0)=%2$.2f"), p, 1-p)

# using a "preformatted" logical vector:
x <- c(TRUE, FALSE, FALSE, NA, TRUE, FALSE)
stri_sprintf("%s") %s", letters[seq_along(x)], c("\u2718", "\u2713")[x+1])

# custom NA/Inf/NaN strings:
stri_printf("%+10.3f", c(-Inf, -0, 0, Inf, NaN, NA_real_),
  na_string="<NA>", nan_string="\U0001F4A9", inf_string="\u221E")

stri_printf("UNIX time %1$s is %1$f UNIX time.", Sys.time())

# the following do not work in sprintf()
stri_sprintf("%1$- *2$.3$f", 1.23456, 10, 3) # two asterisks
stri_sprintf(c("%s", "%f"), pi) # re-coercion needed
stri_sprintf("%1$s is %1$f UNIX time.", Sys.time()) # re-coercion needed
stri_sprintf(c("%d", "%s"), factor(11:12)) # re-coercion needed
stri_sprintf(c("%s", "%d"), factor(11:12)) # re-coercion needed

stri_starts_xwith

Determine if the Start or End of a String Matches a Pattern
Description

These functions check if a string starts or ends with a match to a given pattern. Also, it is possible to check if there is a match at a specific position.

Usage

`stri_startswith(str, ..., fixed, coll, charclass)`

`stri_endswith(str, ..., fixed, coll, charclass)`

`stri_startswith_fixed(  
  str,  
  pattern,  
  from = 1L,  
  negate = FALSE,  
  ...,  
  opts_fixed = NULL  
)`

`stri_endswith_fixed(  
  str,  
  pattern,  
  to = -1L,  
  negate = FALSE,  
  ...,  
  opts_fixed = NULL  
)`

`stri_startswith_charclass(str, pattern, from = 1L, negate = FALSE)`

`stri_endswith_charclass(str, pattern, to = -1L, negate = FALSE)`

`stri_startswith_coll(  
  str,  
  pattern,  
  from = 1L,  
  negate = FALSE,  
  ...,  
  opts_collator = NULL  
)`

`stri_endswith_coll(  
  str,  
  pattern,  
  to = -1L,  
  negate = FALSE,  
  ...,  
  opts_collator = NULL  
)`
### stri_startswith

)  

**Arguments**

- **str** character vector
- **pattern, fixed, coll, charclass** character vector defining search patterns; for more details refer to stringi-search
- **from** integer vector
- **negate** single logical value; whether a no-match to a pattern is rather of interest
- **to** integer vector
- **opts_collator, opts_fixed** a named list used to tune up the search engine’s settings; see stri_opts_collator and stri_opts_fixed, respectively; NULL for the defaults

**Details**

Vectorized over str, pattern, and from or to (with recycling of the elements in the shorter vector if necessary).

If pattern is empty, then the result is NA and a warning is generated.

Argument start controls the start position in str where there is a match to a pattern. to gives the end position.

Indexes given by from or to are of course 1-based, i.e., an index 1 denotes the first character in a string. This gives a typical R look-and-feel.

For negative indexes in from or to, counting starts at the end of the string. For instance, index -1 denotes the last code point in the string.

If you wish to test for a pattern match at an arbitrary position in str, use stri_detect.

stri_startswith and stri_endswith are convenience functions. They call either stri_*_fixed, stri_*_coll, or stri_*_charclass, depending on the argument used. Relying on these underlying functions directly will make your code run slightly faster.

Note that testing for a pattern match at the start or end of a string has not been implemented separately for regex patterns. For that you may use the ‘^’ and ‘$’ meta-characters, see stringi-search-regex.

**Value**

Each function returns a logical vector.

**Author(s)**

Marek Gagolewski and other contributors
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other search_detect: about_search, stri_detect()

Examples

stri_startswith_charclass(' trim me! ','\p{WSpace}')
stri_startswith_fixed(c('a1', 'a2', 'b3', 'a4', 'c5'), 'a')
stri_detect_regex(c('a1', 'a2', 'b3', 'a4', 'c5'), '^a')
stri_startswith_fixed('ababa', 'ba')
stri_startswith_fixed('ababa', 'ba', from=2)
stri_startswith_coll(c('a1', 'A2', 'b3', 'A4', 'C5'), 'a', strength=1)
pat <- stri_paste(\u0635\u0644\u0649 \u0627\u0644\u0644\u0647
\u0639\u0644\u064a\u0647 \u0648\u0633\u0644\u0645XYZ',
\u0639\u0644\u064a\u0647 \u0648\u0633\u0644\u0645XYZ')
stri_endswith_coll('\ufdfa\ufdfa\ufdfaXYZ', pat, strength=1)

stri_stats_general  General Statistics for a Character Vector

Description

This function gives general statistics for a character vector, e.g., obtained by loading a text file with the readLines or stri_read_lines function, where each text line is represented by a separate string.

Usage

stri_stats_general(str)

Arguments

str  character vector to be aggregated

Details

None of the strings may contain \r or \n characters, otherwise you will get at error.

Below by ‘white space‘ we mean the Unicode binary property WHITE_SPACE, see stringi-search-charclass.
Value

Returns an integer vector with the following named elements:

1. Lines - number of lines (number of non-missing strings in the vector);
2. LinesNEmpty - number of lines with at least one non-WHITE_SPACE character;
3. Chars - total number of Unicode code points detected;
4. CharsNWhite - number of Unicode code points that are not WHITE_SPACES;
5. ... (Other stuff that may appear in future releases of stringi).

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other stats: stri_stats_latex()

Examples

```r
s <- c('Lorem ipsum dolor sit amet, consectetur adipisicing elit.',
       'nibh augue, suscipit a, scelerisque sed, lacinia in, mi.',
       'Cras vel lorem. Etiam pellentesque aliquet tellus.','','

stri_stats_general(s)
```

---

<table>
<thead>
<tr>
<th>stri_stats_latex</th>
<th>Statistics for a Character Vector Containing LaTeX Commands</th>
</tr>
</thead>
</table>

Description

This function gives LaTeX-oriented statistics for a character vector, e.g., obtained by loading a text file with the readLines function, where each text line is represented by a separate string.

Usage

```r
stri_stats_latex(str)
```

Arguments

- `str` character vector to be aggregated
stri_sub

Extract a Substring From or Replace a Substring In a Character Vector

Description
stri_sub extracts particular substrings at code point-based index ranges provided. Its replacement version allows to substitute (in-place) parts of a string with given replacement strings. stri_sub_replace is its forward pipe operator-friendly variant that returns a copy of the input vector.

For extracting/replacing multiple substrings from/within each string, see stri_sub_all.

Details
We use a slightly modified LaTeX Word Count algorithm implemented in Kile 2.1.3, see https://kile.sourceforge.io/team.php for the original contributors.

Value
Returns an integer vector with the following named elements:
1. CharsWord - number of word characters;
2. CharsCmdEnvir - command and words characters;
3. CharsWhite - LaTeX white spaces, including { and } in some contexts;
4. Words - number of words;
5. Cmds - number of commands;
6. Envirs - number of environments;
7. ... (Other stuff that may appear in future releases of stringi).

Author(s)
Marek Gagolewski and other contributors

See Also
The official online manual of stringi at https://stringi.gagolewski.com/
Other stats: stri_stats_general()

Examples
s <- c(‘Lorem \textbf{ipsum} dolor sit \textit{amet}, consectetur adipisicing elit.’,’\begin{small}Proin nibh augue,\end{small} suscipit a, scelerisque sed, lacinia in, mi.’,’)
stri_stats_latex(s)
Usage

\[
\text{stri_sub}( \\
\quad \text{str}, \\
\quad \text{from} = 1L, \\
\quad \text{to} = -1L, \\
\quad \text{length}, \\
\quad \text{use\_matrix} = \text{TRUE}, \\
\quad \text{ignore\_negative\_length} = \text{FALSE} \\
) \\
\]

\[
\text{stri_sub}(\text{str}, \text{from} = 1L, \text{to} = -1L, \text{length}, \text{omit\_na} = \text{FALSE}, \text{use\_matrix} = \text{TRUE}) \leftarrow \text{value} \\
\text{stri\_sub\_replace}(\ldots, \text{replacement}, \text{value} = \text{replacement})
\]

Arguments

- **str**: character vector
- **from**: integer vector giving the start indexes; alternatively, if **use_matrix**=TRUE, a two-column matrix of type `cbind(from, to)` (unnamed columns or the 2nd column named other than `length`) or `cbind(from, length=length)` (2nd column named `length`)
- **to**: integer vector giving the end indexes; mutually exclusive with `length` and `from` being a matrix
- **length**: integer vector giving the substring lengths; mutually exclusive with `to` and `from` being a matrix
- **use_matrix**: single logical value; see from
- **ignore_negative_length**: single logical value; whether negative lengths should be ignored or result in missing values
- **omit_na**: single logical value; indicates whether missing values in any of the indexes or in `value` leave the corresponding input string unchanged [replacement function only]
- **value**: a character vector defining the replacement strings [replacement function only]
- **...**: arguments to be passed to `stri_sub<-`
- **replacement**: alias of `value` [wherever applicable]

Details

Vectorized over `str`, `[value]`, `from` and (`to` or `length`). Parameters `to` and `length` are mutually exclusive.

Indexes are 1-based, i.e., the start of a string is at index 1. For negative indexes in `from` or `to`, counting starts at the end of the string. For instance, index -1 denotes the last code point in the string. Non-positive `length` gives an empty string.

Argument `from` gives the start of a substring to extract. Argument `to` defines the last index of a substring, inclusive. Alternatively, its `length` may be provided.
If from is a two-column matrix, then these two columns are used as from and to, respectively, unless the second column is named length. In such a case anything passed explicitly as to or length is ignored. Such types of index matrices are generated by `stri_locate_first` and `stri_locate_last`. If extraction based on `stri_locate_all` is needed, see `stri_sub_all`.

In `stri_sub`, out-of-bound indexes are silently corrected. If from > to, then an empty string is returned. By default, negative length results in the corresponding output being NA, see `ignore_negative_length`, though.

In `stri_sub<-`, some configurations of indexes may work as substring 'injection' at the front, back, or in middle. Negative length does not alter the corresponding input string.

If both to and length are provided, length has priority over to.

Note that for some Unicode strings, the extracted substrings might not be well-formed, especially if input strings are not normalized (see `stri_trans_nfc`), include byte order marks, Bidirectional text marks, and so on. Handle with care.

Value

`stri_sub` and `stri_sub_replace` return a character vector. `stri_sub<-` changes the str object 'in-place'.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other indexing: `stri_locate_all_boundaries()`, `stri_locate_all()`, `stri_sub_all()`

Examples

```r
s <- c("spam, spam, bacon, and spam", "eggs and spam")
stri_sub(s, from=-4)
stri_sub(s, from=1, length=c(10, 4))
(stri_sub(s, 1, 4) <- 'stringi')

x <- c('12 3456 789', 'abc', '', NA, '667')
stri_sub(x, stri_locate_first_regex(x, '[0-9]+')) # see stri_extract_first
stri_sub(x, stri_locate_last_regex(x, '[0-9]+')) # see stri_extract_last

stri_sub_replace(x, stri_locate_first_regex(x, '[0-9]+'),
                 omission=TRUE, replacement='***') # see stri_replace_first
stri_sub_replace(x, stri_locate_last_regex(x, '[0-9]+'),
                 omission=TRUE, replacement='***') # see stri_replace_last

## Not run: x |> stri_sub_replace(1, 5, replacement='new_substring')
```
Select Elements that Match a Given Pattern

Description

These functions return or modify a sub-vector where there is a match to a given pattern. In other words, they are roughly equivalent (but faster and easier to use) to a call to `str[stri_detect(str, ...)]` or `str[stri_detect(str, ...)] <- value`.

Usage

```r
stri_subset(str, ..., regex, fixed, coll, charclass)
stri_subset(str, ..., regex, fixed, coll, charclass) <- value
stri_subset_fixed(str, pattern, omit_na = FALSE, negate = FALSE, ..., opts_fixed = NULL)
stri_subset_fixed(str, pattern, negate=FALSE, ..., opts_fixed=NULL) <- value
stri_subset_charclass(str, pattern, omit_na = FALSE, negate = FALSE)
stri_subset_charclass(str, pattern, negate=FALSE) <- value
stri_subset_coll(str, pattern, omit_na = FALSE, negate = FALSE, ..., opts_collator = NULL)
stri_subset_coll(str, pattern, negate=FALSE, ..., opts_collator=NULL) <- value
stri_subset_regex(str, pattern, omit_na = FALSE, negate = FALSE, ...)
```
opts_regex = NULL
)

stri_subset_regex(str, pattern, negate=FALSE, ..., opts_regex=NULL) <- value

Arguments

str character vector; strings to search within
...
value non-empty character vector of replacement strings; replacement function only
pattern, regex, fixed, coll, charclass character vector; search patterns (no more than the length of str); for more details refer to stringi-search
omit_na single logical value; should missing values be excluded from the result?
negate single logical value; whether a no-match is rather of interest
opts_collator, opts_fixed, opts_regex a named list used to tune up the search engine's settings; see stri_opts_collator, stri_opts_fixed, and stri_opts_regex, respectively; NULL for the defaults

Details

Vectorized over str as well as partially over pattern and value, with recycling of the elements in the shorter vector if necessary. As the aim here is to subset str, pattern cannot be longer than the former. Moreover, if the number of items to replace is not a multiple of length of value, a warning is emitted and the unused elements are ignored. Hence, the length of the output will be the same as length of str.

stri_subset and stri_subset<- are convenience functions. They call either stri_subset_regex, stri_subset_fixed, stri_subset_coll, or stri_subset_charclass, depending on the argument used.

Value

The stri_subset_* functions return a character vector. As usual, the output encoding is UTF-8.

The stri_subset_*<- functions modifies str 'in-place'.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/
Other search_subset: about_search
Examples

```r
stri_subset_regex(c('stringi R', '123', 'ID456', ''), '^[0-9]+$')

x <- c('stringi R', '123', 'ID456', '')
'stri_subset_regex<-'(x, '[0-9]+$', negate=TRUE, value=NA) # returns a copy
stri_subset_regex(x, '[0-9]+$') <- NA # modifies `x` in-place
print(x)
```

---

**stri_sub_all** | *Extract or Replace Multiple Substrings*

### Description

`stri_sub_all` extracts multiple substrings from each string. Its replacement version substitutes (in-place) multiple substrings with the corresponding replacement strings. `stri_sub_replace_all` (alias `stri_sub_all_replace`) is its forward pipe operator-friendly variant, returning a copy of the input vector.

For extracting/replacing single substrings from/within each string, see `stri_sub`.

### Usage

```r
stri_sub_all(
  str,
  from = list(1L),
  to = list(-1L),
  length,
  use_matrix = TRUE,
  ignore_negative_length = TRUE
)
```

```r
stri_sub_all(
  str,
  from = list(1L),
  to = list(-1L),
  length,
  omit_na = FALSE,
  use_matrix = TRUE
) <- value
```

```r
stri_sub_replace_all(..., replacement, value = replacement)
stri_sub_all_replace(..., replacement, value = replacement)
```
Arguments

str character vector
from list of integer vector giving the start indexes; alternatively, if use_matrix=TRUE, a list of two-column matrices of type cbind(from, to) (unnamed columns or the 2nd column named other than length) or cbind(from, length=length) (2nd column named length)
to list of integer vectors giving the end indexes
length list of integer vectors giving the substring lengths
use_matrix single logical value; see from
ignore_negative_length single logical value; whether negative lengths should be ignored or result in missing values
omit_na single logical value; indicates whether missing values in any of the indexes or in value leave the part of the corresponding input string unchanged [replacement function only]
value a list of character vectors defining the replacement strings [replacement function only]
... arguments to be passed to stri_sub_all<-replacement
replacement alias of value [wherever applicable]

Details

Vectorized over str, [value], from and (to or length). Just like in stri_sub, parameters to and length are mutually exclusive.

In one of the simplest scenarios, stri_sub_all(str, from, to), the i-th element of the resulting list generated like stri_sub(str[i], from[[i]], to[[i]]). As usual, if one of the inputs is shorter than the others, recycling rule is applied.

If any of from, to, length, or value is not a list, it is wrapped into a list.

If from consists of a two-column matrix, then these two columns are used as from and to, respectively, unless the second column is named length. Such types of index matrices are generated by stri_locate_all. If extraction or replacement based on stri_locate_first or stri_locate_last is needed, see stri_sub.

In the replacement function, the index ranges must be sorted with respect to from and must be mutually disjoint. Negative length does not result in any altering of the corresponding input string. On the other hand, in stri_sub_all, this make the corresponding chunk be ignored, see ignore_negative_length, though.

Value

stri_sub_all returns a list of character vectors. Its replacement versions modify the input 'in-place'.

Author(s)

Marek Gagolewski and other contributors
See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other indexing: stri_locate_all_boundaries(), stri_locate_all(), stri_sub()

Examples

```r
x <- c(12 345 6789', 'abc', '', NA, '667')
stri_sub_all(x, stri_locate_all_regex(x, '[0-9]+')) # see stri_extract_all
stri_sub_all(x, stri_locate_all_regex(x, '[0-9]+', omit_no_match=TRUE))

stri_sub_all(x, stri_locate_all_regex(x, '[0-9]+', omit_no_match=TRUE)) <- '***'
print(x)

stri_sub_replace_all('a b c', c(1, 3, 5), c(1, 3, 5), replacement=c('A', 'B', 'C'))
```

---

**stri_timezone_get**

*Set or Get Default Time Zone in stringi*

Description

stri_timezone_set changes the current default time zone for all functions in the stringi package, i.e., establishes the meaning of the "NULL time zone" argument to date/time processing functions.

stri_timezone_get gets the current default time zone.

For more information on time zone representation in ICU and stringi, refer to stri_timezone_list.

Usage

```r
stri_timezone_get()
stri_timezone_set(tz)
```

Arguments

- **tz**
  
  single string; time zone identifier

Details

Unless the default time zone has already been set using stri_timezone_set, the default time zone is determined by querying the OS with methods in ICU's internal platform utilities.

Value

stri_timezone_set returns a string with previously used timezone, invisibly.
stri_timezone_get returns a single string with the current default time zone.


**Author(s)**

Marek Gagolewski and other contributors

**References**

`TimeZone` class – ICU API Documentation, [https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1TimeZone.html](https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1TimeZone.html)

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other datetime: `stri_datetime_add()`, `stri_datetime_create()`, `stri_datetime_fields()`, `stri_datetime_format()`, `stri_datetime_fstr()`, `stri_datetime_now()`, `stri_datetime_symbols()`, `stri_timezone_info()`, `stri_timezone_list()`

Other timezone: `stri_timezone_info()`, `stri_timezone_list()`

**Examples**

```r
## Not run:
oldtz <- stri_timezone_set("Europe/Warsaw")
# ... many time zone-dependent operations
stri_timezone_set(oldtz) # restore previous default time zone

## End(Not run)
```

---

**stri_timezone_info**

Query a Given Time Zone

**Description**

Provides some basic information on a given time zone identifier.

**Usage**

`stri_timezone_info(tz = NULL, locale = NULL, display_type = ”long”)`

**Arguments**

- `tz`: NULL or '' for default time zone, or a single string with time zone ID otherwise
- `locale`: NULL or '' for default locale, or a single string with locale identifier
- `display_type`: single string; one of 'short', 'long', 'generic_short', 'generic_long', 'gmt_short', 'gmt_long', 'common', 'generic_location'
**stri_timezone_info**

**Details**

Used to fetch basic information on any supported time zone.

For more information on time zone representation in ICU, see `stri_timezone_list`.

**Value**

Returns a list with the following named components:

1. **ID** (time zone identifier),
2. **Name** (localized human-readable time zone name),
3. **Name.Daylight** (localized human-readable time zone name when DST is used, if available),
4. **Name.Windows** (Windows time zone ID, if available),
5. **RawOffset** (raw GMT offset, in hours, before taking daylight savings into account), and
6. **UsesDaylightTime** (states whether a time zone uses daylight savings time in the current Gregorian calendar year).

**Author(s)**

Marek Gagolewski and other contributors

**See Also**

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other datetime: `stri_datetime_add()`, `stri_datetime_create()`, `stri_datetime_fields()`, `stri_datetime_format()`, `stri_datetime_fstr()`, `stri_datetime_now()`, `stri_datetime_symbols()`, `stri_timezone_get()`, `stri_timezone_list()`

Other timezone: `stri_timezone_get()`, `stri_timezone_list()`

**Examples**

```r
stri_timezone_info()
stri_timezone_info(locale='sk_SK')
sapply(c('short', 'long', 'generic_short', 'generic_long',
       'gmt_short', 'gmt_long', 'common', 'generic_location'),
       function(e) stri_timezone_info('Europe/London', display_type=e))
```
stri_timezone_list  List Available Time Zone Identifiers

Description
Returns a list of available time zone identifiers.

Usage
stri_timezone_list(region = NA_character_, offset = NA_integer_)

Arguments
  region  single string; a ISO 3166 two-letter country code or UN M.49 three-digit area code; NA for all regions
  offset  single numeric value; a given raw offset from GMT, in hours; NA for all offsets

Details
If offset and region are NA (the default), then all time zones are returned. Otherwise, only time zone identifiers with a given raw offset from GMT and/or time zones corresponding to a given region are provided. Note that the effect of daylight savings time is ignored.

A time zone represents an offset applied to the Greenwich Mean Time (GMT) to obtain local time (Universal Coordinated Time, or UTC, is similar, but not precisely identical, to GMT; in ICU the two terms are used interchangeably since ICU does not concern itself with either leap seconds or historical behavior). The offset might vary throughout the year, if daylight savings time (DST) is used, or might be the same all year long. Typically, regions closer to the equator do not use DST. If DST is in use, then specific rules define the point where the offset changes and the amount by which it changes.

If DST is observed, then three additional bits of information are needed:

1. The precise date and time during the year when DST begins. In the first half of the year it is in the northern hemisphere, and in the second half of the year it is in the southern hemisphere.
2. The precise date and time during the year when DST ends. In the first half of the year it is in the southern hemisphere, and in the second half of the year it is in the northern hemisphere.
3. The amount by which the GMT offset changes when DST is in effect. This is almost always one hour.

Value
Returns a character vector.

Author(s)
Marek Gagolewski and other contributors
References

*TimeZone* class – ICU API Documentation, https://unicode-org.github.io/icu-docs/apidoc/dev/icu4c/classicu_1_1TimeZone.html


See Also

The official online manual of *stringi* at https://stringi.gagolewski.com/


Other timezone: *stri_timezone_get()* , *stri_timezone_info()*

Examples

```r
stri_timezone_list()
stri_timezone_list(offset=1)
stri_timezone_list(offset=5.5)
stri_timezone_list(offset=5.75)
stri_timezone_list(region='PL')
stri_timezone_list(region='US', offset=-10)

# Fetch information on all time zones
do.call(rbind.data.frame,
     lapply(stri_timezone_list(), function(tz) stri_timezone_info(tz)))
```

---

**stri_trans_char**  
*Translate Characters*

**Description**

Translates Unicode code points in each input string.

**Usage**

```r
stri_trans_char(str, pattern, replacement)
```

**Arguments**

- `str` character vector
- `pattern` a single character string providing code points to be translated
- `replacement` a single character string giving translated code points
Details

Vectorized over str and with respect to each code point in pattern and replacement.

If pattern and replacement consist of a different number of code points, then the extra code points in the longer of the two are ignored, with a warning.

If code points in a given pattern are not unique, the last corresponding replacement code point is used.

Time complexity for each string in str is $O(\text{stri_length}(\text{str}) \times \text{stri_length}(\text{pattern}))$.

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other transform: stri_trans_general(), stri_trans_list(), stri_trans_nfc(), stri_trans_tolower()

Examples

```r
stri_trans_char('id.123', '.', '_')
stri_trans_char('babaab', 'ab', '01')
stri_trans_char('GCUACGGAGCUUCGGAGCUAG', 'ACGT', 'TGCA')
```

Description

ICU General transforms provide different ways for processing Unicode text. They are useful in handling a variety of different tasks, including:

- locale-independent upper case, lower case, title case, full/halfwidth conversions,
- normalization,
- hex and character name conversions,
- script to script conversion/transliteration.

Usage

```r
stri_trans_general(str, id, rules = FALSE, forward = TRUE)
```
Arguments

- **str**: character vector
- **id**: a single string with transform identifier, see `stri_trans_list`, or custom transliteration rules
- **rules**: if TRUE, treat id as a string with semicolon-separated transliteration rules (see the ICU manual);
- **forward**: transliteration direction (TRUE for forward, FALSE for reverse)

Details

ICU Transforms were mainly designed to transliterate characters from one script to another (for example, from Greek to Latin, or Japanese Katakana to Latin). However, these services are also capable of handling a much broader range of tasks. In particular, the Transforms include prebuilt transformations for case conversions, for normalization conversions, for the removal of given characters, and also for a variety of language and script transliterations. Transforms can be chained together to perform a series of operations and each step of the process can use a UnicodeSet to restrict the characters that are affected.

To get the list of available transforms, call `stri_trans_list`.

Note that transliterators are often combined in sequence to achieve a desired transformation. This is analogous to the composition of mathematical functions. For example, given a script that converts lowercase ASCII characters from Latin script to Katakana script, it is convenient to first (1) separate input base characters and accents, and then (2) convert uppercase to lowercase. To achieve this, a compound transform can be specified as follows: `NFKD; Lower; Latin-Katakana;` (with the default `rules=FALSE`).

Custom rule-based transliteration is also supported, see the ICU manual and below for some examples.

Transliteration is not dependent on the current locale.

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)

Gagolewski M., **stringi**: Fast and portable character string processing in R, *Journal of Statistical Software* 103(2), 2022, 1-59, doi:10.18637/jss.v103.i02

Other transform: `stri_trans_char()`, `stri_trans_list()`, `stri_trans_nfc()`, `stri_trans_tolower()`
Examples

stri_trans_general('gro\u00df', 'latin-ascii')
stri_trans_general('stringi', 'latin-greek')
stri_trans_general('stringi', 'latin-cyrillic')
stri_trans_general('stringi', 'upper') # see stri_trans_toupper
stri_trans_general('\u0104', 'nfd; lower') # compound id; see stri_trans_nfd
stri_trans_general('Marek G\u0105golewski', 'pl-pl_FONIPA')
stri_trans_general('\N{latin small letter a}', 'name-any') # decode name
stri_trans_general('\u2620', 'hex/c') # to hex
stri_trans_general("\u201C\u2026\u201D", "NFKD; NFC; [[^\p{L}]] latin-ascii")

x <- "\u2885\uB85C\uAD6C \uC0AC직동"
stringi::stri_trans_general(x, "Hangul-Latin")
# Deviate from the ICU rules of romanisation of Korean,
# see https://en.wikipedia.org/wiki/Romanization_of_Korean
id <- "
:: NFD;
\u11AB > k;
\u11AE > t;
\u11BB > p;
\u1105 > r;
:: Hangul-Latin;
"

stringi::stri_trans_general(x, id, rules=TRUE)

---

stri_trans_list  List Available Text Transforms and Transliterator

Description

Returns a list of available text transform identifiers. Each of them may be used in stri_trans_general tasks.

Usage

stri_trans_list()

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors
References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other transform: stri_trans_char(), stri_trans_general(), stri_trans_nfc(), stri_trans_tolower()

Examples

stri_trans_list()

stri_trans_nfc

Perform or Check For Unicode Normalization

Description

These functions convert strings to NFC, NFKC, NFD, NFKD, or NFKC_Casefold Unicode Normalization Form or check whether strings are normalized.

Usage

stri_trans_nfc(str)
stri_trans_nfd(str)
stri_trans_nfkd(str)
stri_trans_nfkc(str)
stri_trans_nfkc_casefold(str)
stri_trans_isnfc(str)
stri_trans_isnfd(str)
stri_trans_isnfkd(str)
stri_trans_isnfkc(str)
stri_trans_isnfkc_casefold(str)
**Arguments**

```
str character vector to be encoded
```

**Details**

Unicode Normalization Forms are formally defined normalizations of Unicode strings which, e.g., make possible to determine whether any two strings are equivalent. Essentially, the Unicode Normalization Algorithm puts all combining marks in a specified order, and uses rules for decomposition and composition to transform each string into one of the Unicode Normalization Forms.

The following Normalization Forms (NFs) are supported:

- NFC (Canonical Decomposition, followed by Canonical Composition),
- NFD (Canonical Decomposition),
- NFKC (Compatibility Decomposition, followed by Canonical Composition),
- NFKD (Compatibility Decomposition),
- NFKC_Casefold (combination of NFKC, case folding, and removing ignorable characters which was introduced with Unicode 5.2).

Note that many W3C Specifications recommend using NFC for all content, because this form avoids potential interoperability problems arising from the use of canonically equivalent, yet different, character sequences in document formats on the Web. Thus, you will rather not use these functions in typical string processing activities. Most often you may assume that a string is in NFC, see RFC5198.

As usual in `stringi`, if the input character vector is in the native encoding, it will be automatically converted to UTF-8.

For more general text transforms refer to `stri_trans_general`.

**Value**

The `stri_trans_nf*` functions return a character vector of the same length as input (the output is always in UTF-8).

`stri_trans_isnf*` return a logical vector.

**Author(s)**

Marek Gagolewski and other contributors

**References**

- *Unicode Normalization Forms* – Unicode Standard Annex #15, [https://unicode.org/reports/tr15/](https://unicode.org/reports/tr15/)
- *Character Model for the World Wide Web 1.0: Normalization* – W3C Working Draft, [https://www.w3.org/TR/charmod-norm/](https://www.w3.org/TR/charmod-norm/)
See Also

The official online manual of `stringi` at https://stringi.gagolewski.com/


Other transform: `stri_trans_char()`, `stri_trans_general()`, `stri_trans_list()`, `stri_trans_tolower()`

Examples

```r
stri_trans_nfd('łVar') # a with ogonek -> a, ogonek
stri_trans_nfkc('łVar') # 1 codepoint -> 18 codepoints
```

---

`stri_trans_tolower`  Transform Strings with Case Mapping or Folding

Description

These functions transform strings either to lower case, UPPER CASE, or Title Case or perform case folding.

Usage

```r
stri_trans_tolower(str, locale = NULL)
stri_trans_toupper(str, locale = NULL)
stri_trans_casefold(str)
stri_trans_totitle(str, ..., opts_brkiter = NULL)
```

Arguments

- `str` character vector
- `locale` NULL or `''` for case mapping following the conventions of the default locale, or a single string with locale identifier, see `stringi-locale`.
- `...` additional settings for `opts_brkiter`
- `opts_brkiter` a named list with ICU BreakIterator's settings, see `stri_opts_brkiter`; NULL for default break iterator, i.e., word; `stri_trans_totitle` only

Details

Vectorized over `str`.

ICU implements full Unicode string case mappings. It is worth noting that, generally, case mapping:

- can change the number of code points and/or code units of a string,
• is language-sensitive (results may differ depending on the locale), and
• is context-sensitive (a character in the input string may map differently depending on surrounding characters).

With `stri_trans_totitle`, if word BreakIterator is used (the default), then the first letter of each word will be capitalized and the rest will be transformed to lower case. With the break iterator of type sentence, the first letter of each sentence will be capitalized only. Note that according the ICU User Guide, the string 'one. two. three.' consists of one sentence.

Case folding, on the other hand, is locale-independent. Its purpose is to make two pieces of text that differ only in case identical. This may come in handy when comparing strings.

For more general (but not locale dependent) text transforms refer to `stri_trans_general`.

**Value**
Each function returns a character vector.

**Author(s)**
Marek Gagolewski and other contributors

**References**

**See Also**
The official online manual of stringi at https://stringi.gagolewski.com/


Other locale_sensitive: `%s<%()`, `about_locale`, `about_search_boundaries`, `about_search_coll`, `stri_compare()`, `stri_count_boundaries()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_collator()`, `stri_order()`, `stri_rank()`, `stri_sort_key()`, `stri_sort()`, `stri_split_boundaries()`, `stri_unique()`, `stri_wrap()`

Other transform: `stri_trans_char()`, `stri_trans_general()`, `stri_trans_list()`, `stri_trans_nfc()`

Other text_boundaries: `about_search_boundaries`, `about_search`, `stri_count_boundaries()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_brkiter()`, `stri_split_boundaries()`, `stri_split_lines()`, `stri_wrap()`

**Examples**

```r
stri_trans_toupper("\u00DF", 'de_DE') # small German Eszett / scharfes S
stri_cmp_eq(stri_trans_toupper('i', 'en_US'), stri_trans_toupper('i', 'tr_TR'))
stri_trans_toupper(c('abc', '123', '\u0105\u0104'))
stri_trans_tolower(c('AbC', '123', '\u0105\u0104'))
stri_trans_totitle(c('AbC', '123', '\u0105\u0104'))
stri_trans_casefold(c('AbC', '123', '\u0105\u0104'))
stri_trans_totitle('stringi is a FREE R package. With NO StringS attached.') # word boundary
stri_trans_totitle('stringi is a FREE R package. With NO StringS attached.', type='sentence')
```
**stri_trim_both**  
*Trim Characters from the Left and/or Right Side of a String*

**Description**

These functions may be used, e.g., to remove unnecessary white-spaces from strings. Trimming ends at the first or starts at the last pattern match.

**Usage**

```r
stri_trim_both(str, pattern = "\P{Wspace}", negate = FALSE)
stri_trim_left(str, pattern = "\P{Wspace}", negate = FALSE)
stri_trim_right(str, pattern = "\P{Wspace}", negate = FALSE)
stri_trim(
  str,
  side = c("both", "left", "right"),
  pattern = "\P{Wspace}",
  negate = FALSE
)
```

**Arguments**

- `str` a character vector of strings to be trimmed
- `pattern` a single pattern, specifying the class of characters (see stringi-search-charclass) to be preserved (if `negate` is `FALSE`; default) or trimmed (otherwise)
- `negate` either `TRUE` or `FALSE`; see `pattern`
- `side` character [stri_trim_only]; defaults to 'both'

**Details**

Vectorized over `str` and `pattern`. `stri_trim` is a convenience wrapper over `stri_trim_left` and `stri_trim_right`.

Contrary to many other string processing libraries, our trimming functions are universal. The class of characters to be retained or trimmed can be adjusted.

For replacing pattern matches with an arbitrary replacement string, see `stri_replace`.

Trimming can also be used where you would normally rely on regular expressions. For instance, you may get '23.5' out of 'total of 23.5 bitcoins'.

For trimming white-spaces, please note the difference between Unicode binary property ‘\p{Wspace}’ (more universal) and general character category ‘\p{Z}’, see stringi-search-charclass.

**Value**

All functions return a character vector.
**Description**

Un-escapes all known escape sequences.

**Usage**

```r
stri_unescape_unicode(str)
```

**Arguments**

- `str` character vector

**Details**

Uses ICU's facilities to un-escape Unicode character sequences. The following escape sequences are recognized: \a, \b, \t, \n, \v, \?, \e, \f, \r, \", \', \, \, \uXXXX (4 hex digits), \UXXXXXXXX (8 hex digits), \xXX (1-2 hex digits), \ooo (1-3 octal digits), \cX (control-X; X is masked with 0x1F). For \xXX and \ooo, beware of non-valid UTF-8 byte sequences.

Note that some versions of R on Windows cannot handle characters defined with \UXXXXXXXX.

**Value**

Returns a character vector. If an escape sequence is ill-formed, the result will be NA and a warning will be given.
**stri_unique**

**Extract Unique Elements**

**Description**

This function returns a character vector like `str`, but with duplicate elements removed.

**Usage**

`stri_unique(str, ..., opts_collator = NULL)`

**Arguments**

- `str`: a character vector
- `...`: additional settings for `opts_collator`
- `opts_collator`: a named list with ICU Collator’s options, see `stri_opts_collator`, NULL for default collation options

**Details**

As usual in `stringi`, no attributes are copied. Unlike `unique`, this function tests for canonical equivalence of strings (and not whether the strings are just bytewise equal). Such an operation is locale-dependent. Hence, `stri_unique` is significantly slower (but much better suited for natural language processing) than its base R counterpart.

See also `stri_duplicated` for indicating non-unique elements.

**Value**

Returns a character vector.
Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other locale_sensitive: %s<%(, about_locale, about_search_boundaries, about_search_coll, stri_compare(), stri_count_boundaries(), stri_duplicated(), stri_enc_detect2(), stri_extract_all_boundaries(), stri_locate_all_boundaries(), stri_opts_collator(), stri_order(), stri_rank(), stri_sort_key(), stri_sort(), stri_split_boundaries(), stri_trans_tolower(), stri_wrap()

Examples

# normalized and non-Unicode-normalized version of the same code point:
stri_unique(c("\u0105", stri_trans_nfkd("\u0105")))
unique(c("\u0105", stri_trans_nfkd("\u0105")))
stri_unique(c("gro\u00df", 'GROSS', 'Gro\u00df', 'Gross'), strength=1)

stri_width

Determine the Width of Code Points

Description

Approximates the number of text columns the ‘cat()’ function might use to print a string using a mono-spaced font.

Usage

stri_width(str)

Arguments

str character vector or an object coercible to
Details

The Unicode standard does not formalize the notion of a character width. Roughly based on http://www.cl.cam.ac.uk/~mgk25/ucs/wcwidth.c, https://github.com/nodejs/node/blob/master/src/node_i18n.cc, and UAX #11 we proceed as follows. The following code points are of width 0:

- code points with general category (see stringi-search-charclass) Me, Mn, and Cf,
- C0 and C1 control codes (general category Cc) - for compatibility with the nchar function,
- Hangul Jamo medial vowels and final consonants (code points with enumerable property UCHAR_HANGUL_SYLLABLE_TYPE equal to U_HST_VOWEL_JAMO or U_HST_TRAILING_JAMO; note that applying the NFC normalization with stri_trans_nfc is encouraged),
- ZERO WIDTH SPACE (U+200B),

Characters with the UCHAR_EAST_ASIAN_WIDTH enumerable property equal to U_EA_FULLWIDTH or U_EA_WIDE are of width 2.

Most emojis and characters with general category So (other symbols) are of width 2.

SOFT HYPHEN (U+00AD) (for compatibility with nchar) as well as any other characters have width 1.

Value

Returns an integer vector of the same length as str.

Author(s)

Marek Gagolewski and other contributors

References


See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other length: %s%%(), stri_isempty(), stri_length(), stri_numbytes(), stri_pad_both(), stri_sprintf()

Examples

stri_width(LETTERS[1:5])
stri_width(stri_trans_nfkd(‘\u0105’))
stri_width(stri_trans_nfkd(‘\U0001F606’))
stri_width( # Full-width equivalents of ASCII characters:
  stri_enc_fromutf32(as.list(c(0x3000, 0xFF01:0xFF5E)))
)
stri_width(stri_trans_nfkd(‘\ubc1f’)) # includes Hangul Jamo medial vowels and final consonants
Word Wrap Text to Format Paragraphs

Description

This function breaks text paragraphs into lines, of total width (if it is possible) at most given width.

Usage

```r
stri_wrap(
  str,
  width = floor(0.9 * getOption("width")),
  cost_exponent = 2,
  simplify = TRUE,
  normalize = TRUE,
  indent = 0,
  exdent = 0,
  prefix = "",
  initial = prefix,
  whitespace_only = FALSE,
  use_length = FALSE,
  locale = NULL
)
```

Arguments

- **str**: character vector of strings to reformat
- **width**: single integer giving the suggested maximal total width/number of code points per line
- **cost_exponent**: single numeric value, values not greater than zero will select a greedy word-wrapping algorithm; otherwise this value denotes the exponent in the cost function of a (more aesthetic) dynamic programming-based algorithm (values in [2, 3] are recommended)
- **simplify**: single logical value, see Value
- **normalize**: single logical value, see Details
- **indent**: single non-negative integer; gives the indentation of the first line in each paragraph
- **exdent**: single non-negative integer; specifies the indentation of subsequent lines in paragraphs
- **prefix, initial**: single strings; prefix is used as prefix for each line except the first, for which initial is utilized
whitespace_only: single logical value; allow breaks only at white-spaces? If FALSE, ICU’s line break iterator is used to split text into words, which is suitable for natural language processing.

use_length: single logical value; should the number of code points be used instead of the total code point width (see stri_width)?

locale: NULL or ‘’ for text boundary analysis following the conventions of the default locale, or a single string with locale identifier, see stringi-locale.

Details

Vectorized over str.

If whitespace_only is FALSE, then ICU’s line-breakIterator is used to determine text boundaries where a line break is possible. This is a locale-dependent operation. Otherwise, the breaks are only at white-spaces.

Note that Unicode code points may have various widths when printed on the console and that this function, by default, takes that into account. By changing the state of the use_length argument, this function starts to act as if each code point was of width 1.

If normalize is FALSE, then multiple white spaces between the word boundaries are preserved within each wrapped line. In such a case, none of the strings can contain \r, \n, or other new line characters, otherwise you will get an error. You should split the input text into lines or, for example, substitute line breaks with spaces before applying this function.

If normalize is TRUE, then all consecutive white space (ASCII space, horizontal TAB, CR, LF) sequences are replaced with single ASCII spaces before actual string wrapping. Moreover, stri_split_lines and stri_trans_nfc is called on the input character vector. This is for compatibility with strwrap.

The greedy algorithm (for cost_exponent being non-positive) provides a very simple way for word wrapping. It always puts as many words in each line as possible. This method – contrary to the dynamic algorithm – does not minimize the number of space left at the end of every line. The dynamic algorithm (a.k.a. Knuth’s word wrapping algorithm) is more complex, but it returns text wrapped in a more aesthetic way. This method minimizes the squared (by default, see cost_exponent) number of spaces (raggedness) at the end of each line, so the text is more arranged evenly. Note that the cost of printing the last line is always zero.

Value

If simplify is TRUE, then a character vector is returned. Otherwise, you will get a list of length(str) character vectors.

Author(s)

Marek Gagolewski and other contributors

References

See Also
The official online manual of `stringi` at https://stringi.gagolewski.com/
Other locale_sensitive: `%s<%()`, `about_locale`, `about_search_boundaries`, `about_search_coll`, `stri_compare()`, `stri_count_boundaries()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_collator()`, `stri_order()`, `stri_rank()`, `stri_sort_key()`, `stri_sort()`, `stri_split_boundaries()`, `stri_trans_tolower()`, `stri_unique()`
Other text_boundaries: `about_search_boundaries`, `about_search`, `stri_count_boundaries()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_brkiter()`, `stri_split_boundaries()`, `stri_split_lines()`, `stri_trans_tolower()`

Examples
```r
s <- stri_paste(
  "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin ",
  "nibh augue, suscipit a, scelerisque sed, lacinia in, mi. Cras vel ",
  "lorem. Etiam pellentesque aliquet tellus."
)  
cat(stri_wrap(s, 20, 0.0), sep="\n") # greedy

cat(stri_wrap(s, 20, 2.0), sep="\n") # dynamic

cat(stri_pad(stri_wrap(s), side='both'), sep='\n')
```

---

**stri_write_lines**

*Write Text Lines to a Text File*

**Description**

Writes a text file in such a way that each element of a given character vector becomes a separate text line.

**Usage**

```r
stri_write_lines(
  str,
  con,
  encoding = "UTF-8",
  sep = ifelse(.Platform$OS.type == "windows", "\r\n", "\n"),
  fname = con
)
```

**Arguments**

- `str` character vector with data to write
- `con` name of the output file or a connection object (opened in the binary mode)
- `encoding` output encoding, NULL or `'` for the current default one
- `sep` newline separator
- `fname` [DEPRECATED] alias of `con`
Details

It is a substitute for the R `writeLines` function, with the ability to easily re-encode the output. We suggest using the UTF-8 encoding for all text files: thus, it is the default one for the output.

Value

This function returns nothing noteworthy.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other files: `stri_read_lines()`, `stri_read_raw()`

---

**Concatenate Two Character Vectors**

Description

Binary operators for joining (concatenating) two character vectors, with a typical R look-and-feel.

Usage

```r
e1 %s+% e2
```

```r
e1 %stri+% e2
```

Arguments

- `e1`: a character vector or an object coercible to a character vector
- `e2`: a character vector or an object coercible to a character vector

Details

Vectorized over `e1` and `e2`. These operators act like a call to `stri_join(e1, e2, sep=' ')`. However, note that joining 3 vectors, e.g., `e1 %s+% e2 %s+% e3` is slower than `stri_join(e1, e2, e3, sep=' ')`, because it creates a new (temporary) result vector each time the operator is applied.

Value

Returns a character vector.
Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of stringi at https://stringi.gagolewski.com/


Other join: stri_dup(), stri_flatten(), stri_join_list(), stri_join()

Examples

c('abc', '123', 'xy') %s+% letters[1:6]
'ID_' %s+% 1:5

Description

Relational operators for comparing corresponding strings in two character vectors, with a typical R look-and-feel.

Usage

e1 %s<% e2

e1 %s<=% e2

e1 %s>% e2

e1 %s>=% e2

e1 %s==% e2

e1 %s!==% e2

e1 %stri<% e2

e1 %stri<=% e2
e1 %stri>% e2
e1 %stri>=% e2
e1 %stri==% e2
e1 %stri!=% e2
e1 %stri===% e2
e1 %stri!==% e2

Arguments

e1, e2 character vectors or objects coercible to character vectors

Details

These functions call `stri_cmp_le` or its friends, using the default collator options. As a consequence, they are vectorized over `e1` and `e2`.

%stri==% tests for canonical equivalence of strings (see `stri_cmp_equiv`) and is a locale-dependent operation.

%stri===% performs a locale-independent, code point-based comparison.

Value

All the functions return a logical vector indicating the result of a pairwise comparison. As usual, the elements of shorter vectors are recycled if necessary.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other locale_sensitive: `about_locale`, `about_search_boundaries`, `about_search_coll`, `stri_compare()`, `stri_count_boundaries()`, `stri_duplicated()`, `stri_enc_detect2()`, `stri_extract_all_boundaries()`, `stri_locate_all_boundaries()`, `stri_opts_collator()`, `stri_order()`, `stri_rank()`, `stri_sort_key()`, `stri_sort()`, `stri_split_boundaries()`, `stri_trans_tolower()`, `stri_unique()`, `stri_wrap()`

Examples

'a' %stri<% 'b'
c('a', 'b', 'c') %stri>=% 'b'
C-Style Formatting with `stri_sprintf` as a Binary Operator

Description

Provides access to `stri_sprintf` in form of a binary operator in a way similar to Python’s `%` overloaded for strings.

Missing values and empty vectors are propagated as usual.

Usage

\[
e_1 \ %s\ %\ e_2
\]

\[
e_1 \ %\text{stri}\ %\ e_2
\]

Arguments

- \(e_1\) format strings, see `stri_sprintf` for syntax
- \(e_2\) a list of atomic vectors to be passed to `stri_sprintf` or a single atomic vector

Details

Vectorized over \(e_1\) and \(e_2\).

\(e_1 \ %s\ %\ \text{atomic\_vector}\) is equivalent to \(e_1 \ %s\ %\ \text{list(atomic\_vector)}\).

Value

Returns a character vector.

Author(s)

Marek Gagolewski and other contributors

See Also

The official online manual of `stringi` at [https://stringi.gagolewski.com/](https://stringi.gagolewski.com/)


Other length: `stri_isempty()`, `stri_length()`, `stri_numbytes()`, `stri_pad_both()`, `stri_sprintf()`, `stri_width()`
Examples

"value=%d" %s% 3
"value=%d" %s% 1:3
"%s=%d" %s% list("value", 3)
"%s=%d" %s% list("value", 1:3)
"%s=%d" %s% list(c("a", "b", "c"), 1)
"%s=%d" %s% list(c("a", "b", "c"), 1:3)

x <- c("abcd", "\u00DF\u00B5\U0001F970", "abcdef")
cat("[%6s] %s% x, sep="\n") # width used, not the number of bytes
Index

* datetime
  stri_datetime_add, 33
  stri_datetime_create, 34
  stri_datetime_fields, 36
  stri_datetime_format, 37
  stri_datetime_fstr, 41
  stri_datetime_now, 42
  stri_datetime_symbols, 43
  stri_timezone_get, 143
  stri_timezone_info, 144
  stri_timezone_list, 146
* encoding_conversion
  about_encoding, 5
  stri_enc_fromutf32, 55
  stri_enc_toascii, 63
  stri_enc_tonative, 64
  stri_enc_toutf32, 65
  stri_enc_toutf8, 66
  stri_encode, 50
* encoding_detection
  about_encoding, 5
  stri_enc_detect, 52
  stri_enc_detect2, 54
  stri_enc_isascii, 57
  stri_enc_isutf16be, 58
  stri_enc_isutf8, 59
* encoding_management
  about_encoding, 5
  stri_enc_info, 56
  stri_enc_list, 60
  stri_enc_mark, 61
  stri_enc_set, 62
* encoding
  stri_info, 75
* escape
  stri_escape_unicode, 67
  stri_unescape_unicode, 156
* files
  stri_read_lines, 111
  stri_read_raw, 112
  stri_write_lines, 162
* indexing
  stri_locate_all, 85
  stri_locate_all_boundaries, 89
  stri_sub, 136
  stri_sub_all, 141
* join
  %s+, 163
  stri_dup, 47
  stri_flatten, 74
  stri_join, 77
  stri_join_list, 78
* length
  %s%, 166
  stri_isempty, 76
  stri_length, 79
  stri_numbytes, 95
  stri_pad_both, 104
  stri_sprintf, 129
  stri_width, 158
* locale_management
  about_locale, 9
  stri_locale_info, 82
  stri_locale_list, 83
  stri_locale_set, 84
* locale_sensitive
  %s<, 164
  about_locale, 9
  about_search_boundaries, 12
  about_search_coll, 19
  stri_compare, 27
  stri_count_boundaries, 31
  stri_duplicated, 48
  stri_enc_detect2, 54
  stri_extract_all_boundaries, 72
  stri_locate_all_boundaries, 89
  stri_opts_collator, 98
  stri_order, 103

168
stri_rank, 109
stri_sort, 120
stri_sort_key, 121
stri_split_boundaries, 125
stri_trans_tolower, 153
stri_unique, 157
stri_wrap, 160

* locale
  stri_info, 75

* prepare_arg
  about_arguments, 4

* random
  stri_rand_lipsum, 106
  stri_rand_shuffle, 107
  stri_rand_strings, 108

* search_charclass
  about_search, 11
  about_search_charclass, 14
  stri_trim_both, 155

* search_coll
  about_search, 11
  about_search_coll, 19
  stri_opts_collator, 98

* search_count
  about_search, 11
  stri_count, 29
  stri_count_boundaries, 31

* search_detect
  about_search, 11
  stri_detect, 45
  stri_startswith, 131

* search_extract
  about_search, 11
  stri_extract_all, 68
  stri_extract_all_boundaries, 72
  stri_match_all, 91

* search_fixed
  about_search, 11
  about_search_fixed, 20
  stri_opts_fixed, 100

* search_in
  about_search, 11

* search_locate
  about_search, 11
  stri_locate_all, 85
  stri_locate_all_boundaries, 89

* search_regex
  about_search, 11

  about_search_regex, 21
  stri_opts_regex, 101

* search_replace
  about_search, 11
  stri_replace_all, 114
  stri_replace_rstr, 118
  stri_trim_both, 155

* search_split
  about_search, 11
  stri_split, 122
  stri_split_boundaries, 125
  stri_split_lines, 127

* search_subset
  about_search, 11
  stri_subset, 139

* stats
  stri_stats_general, 134
  stri_stats_latex, 135

* stringi_general_topics
  about_arguments, 4
  about_encoding, 5
  about_locale, 9
  about_search, 11
  about_search_boundaries, 12
  about_search_charclass, 14
  about_search_coll, 19
  about_search_fixed, 20
  about_search_regex, 21
  about_stringi, 25

* text_boundaries
  about_search, 11
  about_search_boundaries, 12
  stri_count_boundaries, 31
  stri_extract_all_boundaries, 72
  stri_locate_all_boundaries, 89
  stri_opts_brkiter, 96
  stri_split_boundaries, 125
  stri_split_lines, 127
  stri_trans_tolower, 153
  stri_wrap, 160

* timezone
  stri_timezone_get, 143
  stri_timezone_info, 144
  stri_timezone_list, 146

* transform
  stri_trans_char, 147
  stri_trans_general, 148
  stri_trans_list, 150


about_search_charclass, 5, 8, 10–13, 14, 20, 21, 24–27, 156
about_search_coll, 5, 8, 10–13, 19, 19, 20, 21, 24, 25, 27, 28, 32, 49, 55, 73, 91, 99, 104, 110, 121, 122, 127, 154, 158, 162, 165
about_search_fixed, 5, 8, 10–13, 19, 20, 24, 25, 27, 101
about_search_regex, 5, 8, 10–14, 19–21, 21, 25, 27, 103
about_stringi, 5, 8, 10, 12, 13, 19–21, 25, 25
anyDuplicated, 49
arguments (about_arguments), 4
as.character, 4
as.list, 82
cat, 5, 7, 130
dim, 5
duplicated, 49
eenc2utf8, 66
Encoding, 7, 58–60, 62
encoding (about_encoding), 5
gsub, 118
iconv, 50
intToUtf8, 56
locale (about_locale), 9
names, 5
nchar, 159
oper_compare (%s<%), 164
oper_comparison (%s<%), 164
oper_dollar (%s$), 166
oper_multiply (stri_dup), 47
oper_plus (%s+), 163
operator_add (%s+), 163
operator_compare (%s<%), 164
operator_dollar (%s$), 166
operator_multiply (stri_dup), 47
operator_plus (%s+), 163
paste, 74, 77
POSIXct, 33, 35, 36, 38, 41, 43
print, 5, 7
Quotes, 67
rawToChar, 51
readLines, 111, 134, 135
regex, 21

search (about_search), 11
search_boundaries
  (about_search_boundaries), 12
search_charclass
  (about_search_charclass), 14
search_coll (about_search_coll), 19
search_fixed (about_search_fixed), 20
search_regex (about_search_regex), 21
simplify2array, 81
sprintf, 129, 130
strftime, 38, 41
stri_c(stri_join), 77
stri_c_list (stri_join_list), 78
stri_cmp, 26
stri_cmp (stri_compare), 27
stri_cmp_eq (stri_compare), 27
stri_cmp_equiv, 165
stri_cmp_ge (stri_compare), 27
stri_cmp_gt (stri_compare), 27
stri_cmp_le, 165
stri_cmp_lt (stri_compare), 27
stri_cmp_neq (stri_compare), 27
stri_cmp_nequiv (stri_compare), 27
stri_coll (stri_opts_collator), 98
stri_compare, 10, 13, 20, 27, 32, 49, 55, 73, 91, 98, 99, 104, 110, 121, 122, 127, 154, 158, 162, 165
stri_conv (stri_encode), 50
stri_count, 11, 12, 29, 32
stri_count_boundaries, 10, 12, 13, 20, 26, 28, 30, 31, 32, 49, 55, 73, 80, 91, 97, 99, 104, 110, 121, 122, 127, 154, 158, 162, 165
stri_count_charclass (stri_count), 29
stri_count_coll (stri_count), 29
stri_count_fixed, 100
stri_count_fixed (stri_count), 29
stri_count_regex, 101
stri_count_regex (stri_count), 29
stri_count_words, 73, 90
stri_count_words
  (stri_count_boundaries), 31

stri_datetime_add, 33, 35, 37, 41–44, 144, 145, 147
stri_datetime_add<- (stri_datetime_add), 33
stri_datetime_create, 34, 35, 37, 41–44, 144, 145, 147
stri_datetime_fields, 34, 35, 36, 41–44, 144, 145, 147
stri_datetime_format, 26, 34, 35, 37, 37, 41–44, 144, 145, 147
stri_datetime_fstr, 34, 35, 37, 38, 41, 43, 44, 144, 145, 147
stri_datetime_now, 34, 35, 37, 41, 42, 42, 44, 144, 145, 147
stri_datetime_parse, 41
stri_datetime_parse
  (stri_datetime_format), 37
stri_datetime_symbols, 34–37, 41–43, 43, 144, 145, 147
stri_detect, 11, 12, 45, 133, 134, 139
stri_detect_charclass (stri_detect), 45
stri_detect_coll, 98
stri_detect_coll (stri Detect), 45
stri_detect_fixed (stri Detect), 45
stri_detect_regex (stri Detect), 45
stri_dup, 26, 47, 74, 78, 79, 164
stri_duplicated, 10, 13, 20, 26, 28, 32, 48, 55, 73, 91, 98, 99, 104, 110, 121, 122, 127, 154, 158, 157, 158, 162, 165
stri_duplicated_any (stri_duplicated), 48
stri_enc_detect, 8, 52, 54, 55, 58–60, 112
stri_enc_detect2, 8, 10, 13, 20, 28, 32, 49, 54, 54, 58–60, 73, 91, 99, 104, 110, 121, 122, 127, 154, 158, 162, 165
stri_enc_fromutf32, 8, 51, 55, 64–67
stri_enc_get, 7, 50, 51, 62, 65, 66
stri_enc_get (stri_enc_set), 62
stri_enc_info, 8, 56, 61–63, 75
stri_enc_isascii, 8, 54, 55, 57, 59, 60
stri_enc_isutf16be, 8, 54, 55, 58, 58, 60
stri_enc_isutf16le
  (stri_enc_isutf16be), 58
stri_enc_isutf32be
  (stri_enc_isutf16be), 58
stri_enc_isutf32le
  (stri_enc_isutf16be), 58
stri_enc_isutf8, 6, 8, 54, 55, 58, 59, 59, 60
INDEX

151, 151, 154, 159, 161
stri_trans_nfd (stri_trans_nfc), 151
stri_trans_nfkc (stri_trans_nfc), 151
stri_trans_nfkc_casefold (stri_trans_nfc), 151
stri_trans_nfkd (stri_trans_nfc), 151
stri_trans_tolower, 10, 12, 13, 20, 26, 28, 32, 49, 55, 73, 91, 97, 99, 104, 110, 121, 122, 127, 128, 148, 149, 151, 153, 158, 162, 165
stri_trans_totitle, 12
stri_trans_totitle (stri_trans_tolower), 153
stri_trans_toupper (stri_trans_tolower), 153
stri_trim, 11, 26, 116
stri_trim (stri_trim_both), 155
stri_trim_both, 12, 19, 116, 119, 155
stri_trim_left, 105
stri_trim_left (stri_trim_both), 155
stri_trim_right (stri_trim_both), 155
stri_unescape_unicode, 68, 156
stri_unique, 10, 12, 13, 20, 26, 28, 32, 49, 55, 73, 91, 98, 99, 104, 110, 121, 122, 127, 154, 157, 162, 165
stri_width, 26, 76, 80, 95, 105, 106, 130, 131, 158, 161, 166
stri_wrap, 10, 12, 13, 20, 26, 28, 32, 49, 55, 73, 91, 97, 99, 104, 105, 110, 121, 122, 127, 128, 154, 158, 160, 165
stri_write_lines, 26, 111, 112, 162
stringi (about_stringi), 25
stringi-arguments (about_arguments), 4
stringi-encoding, 30, 58–63
stringi-encoding (about_encoding), 5
stringi-locale, 31, 72, 82–84, 90, 153, 161
stringi-locale (about_locale), 9
stringi-package (about_stringi), 25
stringi-search, 30, 46, 70, 87, 92, 116, 124, 133, 140
stringi-search (about_search), 11
stringi-search-boundaries, 31, 73, 90, 96, 126
stringi-search-boundaries (about_search_boundaries), 12
stringi-search-charclass, 108, 155, 159
stringi-search-charclass (about_search_charclass), 14
stringi-search-col1, 98, 100
stringi-search-col1 (about_search_col1), 19
stringi-search-fixed, 100, 102
stringi-search-fixed (about_search_fixed), 20
stringi-search-regex, 101, 102, 133
stringi-search-regex (about_search_regex), 21
strptime, 38, 41, 42
strwrap, 161
Sys.getlocale, 7, 62, 63
unique, 157
utf8ToInt, 65
writeLines, 163