Package 'stringi'

May 2, 2018

Version 1.2.2

Date 2018-05-01

Title Character String Processing Facilities

Description Allows for fast, correct, consistent, portable,
as well as convenient character string/text processing in every locale
and any native encoding. Owing to the use of the 'ICU' library,
the package provides 'R' users with platform-independent functions
known to 'Java', 'Perl', 'Python', 'PHP', and 'Ruby' programmers. Available
features include: pattern searching (e.g., with 'Java'-like regular
expressions or the 'Unicode' collation algorithm), random string generation,
case mapping, string transliteration, concatenation,
Unicode normalization, date-time formatting and parsing, and many more.

URL http://www.gagolewski.com/software/stringi/
http://site.icu-project.org/ http://www.unicode.org/

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

SystemRequirements ICU4C (>= 52, optional)

Type Package

Depends R (>= 2.14)

Imports tools, utils, stats

Biarch TRUE

License file LICENSE

Author Marek Gagolewski [aut, cre], Bartek Tartanus [ctb],
and other contributors (stringi source code);
IBM and other contributors (ICU4C source code);
Unicode, Inc. (Unicode Character Database)

Maintainer Marek Gagolewski <gagolews@rexamine.com>

RoxygenNote 6.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-05-02 17:44:33 UTC

R topics documented:

`stringi-package` *THE String Processing Package*

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: `http://www.gagolewski.com/software/stringi/`

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:

- stringi-encoding – character encoding issues, including information on encoding management in **stringi**, as well as on encoding detection and conversion.
- stringi-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, sorting, case-folding, and searching.
- stringi-arguments – information on how **stringi** treats its functions' arguments.

Facilities available

Refer to the following:

- stringi-search for string searching facilities; these include pattern searching, matching, string splitting, and so on. The following independent search engines are provided:
  - stringi-search-regex – with ICU (Java-like) regular expressions,
  - stringi-search-fixed – fast, locale-independent, bytewise pattern matching,
  - stringi-search-coll – locale-aware pattern matching for natural language processing tasks,
  - stringi-search-charclass – seeking elements of particular character classes, like "all whitespaces" or "all digits",
  - stringi-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_join, stri_dup, %s+%, and stri_flatten for concatenation-based operations.
- stri_sub for extracting and replacing substrings, and stri_reverse for a joyful function to reverse all code points in a string.
- 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 stringi-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 invidual code points, and stri_trans_general for other very general yet powerful text transforms, including transliteration.
- stri_cmp, %s<%, stri_order, stri_sort, stri_unique, and stri_duplicated for collation-based, locale-aware operations, see also stringi-locale.
- stri_split_lines (among others) to split a string into text lines.
- stri_escape_unicode (among others) for escaping certain code points.
- stri_rand_strings, stri_rand_shuffle, and stri_rand_lipsum for generating (pseudo)random strings.
- DRAFT API: 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 others. ICU4C was developed by IBM and others. The Unicode Character Database is due to Unicode, Inc.; see the COPYRIGHTS file for more details.

References

stringi *Package homepage*, http://www.gagolewski.com/software/stringi/

ICU – International Components for Unicode, http://www.icu-project.org/

ICU4C API Documentation, http://www.icu-project.org/apiref/icu4c/

The Unicode Consortium, http://www.unicode.org/

UTF-8, a transformation format of ISO 10646 – RFC 3629, http://tools.ietf.org/html/rfc3629

See Also

Other stringi_general_topics: stringi-arguments, stringi-encoding, stringi-locale, stringi-search-boundaries, stringi-search-charclass, stringi-search-coll, stringi-search-fixed, stringi-search-regex, stringi-search

stringi-arguments *Passing Arguments to Functions in* **stringi**

Description

Below we explain how **stringi** deals (in most of the cases) with its functions' arguments.

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 of non-atomic vectors each of length 1 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; This may sometimes lead to strange results - we assume you know what you are doing. However, due to this property you may, e.g., search for one pattern in each given string, or search for each pattern in one given 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 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 Input Objects' Attributes

Generally, all our functions drop input objects' attributes (e.g., `names`, `dim`, etc.). This is generally because of advanced vectorization and for efficiency reasons. Thus, if arguments' preserving is needed, please remember to copy important attributes manually or use, e.g., the subsetting operation like `x[] <- stri_...(x, ...)`.

See Also

Other stringi_general_topics: `stringi-encoding`, `stringi-locale`, `stringi-package`, `stringi-search-boundaries`, `stringi-search-charclass`, `stringi-search-coll`, `stringi-search-fixed`, `stringi-search-regex`, `stringi-search`

stringi-encoding *Character Encodings and* **stringi**

Description

This man page aims to explain (or at least to cast light on) how **stringi** deals with character strings in various encodings.

In particular you should note that:

- R lets strings in ASCII, UTF-8, and your platform's native encoding coexist peacefully. Character vector output with `print`, `cat` etc. silently reencodes each string so that it can be properly shown e.g. in the R's console.
- Functions in **stringi** process each string internally in Unicode, which is a superset of all character representation schemes. 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 doing more complex operations like string comparison, sorting, and even concatenation). Thus, it is best to rely on cascading calls to **stringi** operations solely.

Details

"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).

In most cases, Unicode is a superset of the 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, cf. `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). Thanks 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 (it is for example 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 ups 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

You should keep in mind that data in memory are just bytes (small integer values) – an en*coding* 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.

Basically, R has a very simple encoding marking mechanism, see `stri_enc_mark`. There is an implicit assumption that your platform's default (native) encoding is always a superset of ASCII – **stringi** checks that when 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, macOS, 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 macOS 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 inputting a string e.g. via a keyboard. 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 peacefully. Character vector printed with `print`, `cat` etc. silently reencodes each string 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 load it from a file saved in 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 `stri_enc_toascii` (substitutes all non-ASCII bytes with the SUBSTITUTE CHARACTER, which plays a similar role as R's `NA` value).

There are also some routines for automated encoding detection, see e.g. `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 `stri_enc_detect` (among others) for a useful function in this category.

References

Unicode Basics – ICU User Guide, http://userguide.icu-project.org/unicode

Conversion – ICU User Guide, http://userguide.icu-project.org/conversion

Converters – ICU User Guide, http://userguide.icu-project.org/conversion/converters (technical details)

UTF-8, UTF-16, UTF-32 & BOM – ICU FAQ, http://www.unicode.org/faq/utf_bom.html

See Also

Other stringi_general_topics: stringi-arguments, stringi-locale, stringi-package, stringi-search-boundaries, stringi-search-charclass, stringi-search-coll, stringi-search-fixed, stringi-search-regex, stringi-search

Other encoding_management: stri_enc_info, stri_enc_list, stri_enc_mark, stri_enc_set

Other encoding_detection: stri_enc_detect2, stri_enc_detect, stri_enc_isascii, stri_enc_isutf16be, stri_enc_isutf8

Other encoding_conversion: stri_enc_fromutf32, stri_enc_toascii, stri_enc_tonative, stri_enc_toutf32, stri_enc_toutf8, stri_encode

---

stringi-locale *Locales and* **stringi**

Description

In this section we explain how we deal with 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 case, **ICU** `Variant` "_EURO" could be used for selecting locales that support the Euro currency.

The final (optional) element of a locale is an optional 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`, `colation`, `currency`, and `numbers`, e.g., `fr@collation=phonebook;calendar=islamic-`

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`.

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: they are initially set to be the system locale on that platform, and may be changed with `stri_locale_set`, for example, if automatic detection fails to recognize your locale properly.

Sometimes it is suggested that your program should avoid changing the default locale: it is not a good way to request an international object, especially only for a single function call. 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.

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

References

Locale – ICU User Guide, http://userguide.icu-project.org/locale

ISO 639: Language Codes, http://www.iso.org/iso/home/standards/language_codes.htm

ISO 3166: Country Codes, http://www.iso.org/iso/country_codes

See Also

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

Other locale_sensitive: `%s<%`, `stri_compare`, `stri_count_boundaries`, `stri_duplicated`, `stri_enc_detect2`, `stri_extract_all_boundaries`, `stri_locate_all_boundaries`, `stri_opts_collator`, `stri_order`, `stri_split_boundaries`, `stri_trans_tolower`, `stri_unique`, `stri_wrap`, `stringi-search-boundaries`, `stringi-search-coll`

Other stringi_general_topics: `stringi-arguments`, `stringi-encoding`, `stringi-package`, `stringi-search-boundaries` `stringi-search-charclass`, `stringi-search-coll`, `stringi-search-fixed`, `stringi-search-regex`, `stringi-search`

stringi-search *String Searching*

Description

This man page instructs 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, see stringi-search-regex,
- `stri_*_fixed` – locale-independent bytewise pattern matching, see stringi-search-fixed,
- `stri_*_coll` – **ICU**'s `StringSearch`, locale-sensitive, Collator-based pattern search, useful for natural language processing tasks, see stringi-search-coll,
- `stri_*_charclass` – character classes search, e.g. Unicode General Categories or Binary Properties, see stringi-search-charclass,
- `stri_*_boundaries` – text boundary analysis, see stringi-search-boundaries

Each "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`.

See Also

Other text_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`, `stringi-search-boundaries`

Other search_regex: `stri_opts_regex`, `stringi-search-regex`

Other search_fixed: `stri_opts_fixed`, `stringi-search-fixed`

Other search_coll: `stri_opts_collator`, `stringi-search-coll`

Other search_charclass: stri_trim_both, stringi-search-charclass

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_na, 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: stringi-arguments, stringi-encoding, stringi-locale, stringi-package, stringi-search-boundaries, stringi-search-charclass, stringi-search-coll, stringi-search-fixed, stringi-search-regex

---

`stringi-search-boundaries`

Text Boundary Analysis in **stringi**

Description

Text boundary analysis is the process of locating linguistic boundaries while formatting and handling text.

Details

Examples of the boundary analysis process process include:

- Locating appropriate points to word-wrap text to fit within specific margins while displaying or printing, see stri_wrap and stri_split_boundaries.
- Counting characters, words, sentences, or paragraphs, see stri_count_boundaries.
- Making a list of the unique words in a document, cf. stri_extract_all_words and then stri_unique.
- Capitalizing the first letter of each word or sentence, see also stri_trans_totitle.
- Locating 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 come 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 points to wrap lines when displaying the text.
- On the other hand, a 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.

References

Boundary Analysis – ICU User Guide, http://userguide.icu-project.org/boundaryanalysis

See Also

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-coll

Other text_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, stringi-search

Other stringi_general_topics: stringi-arguments, stringi-encoding, stringi-locale, stringi-package, stringi-search-charclass, stringi-search-coll, stringi-search-fixed, stringi-search-regex, stringi-search

---

stringi-search-charclass

Character Classes in **stringi**

Description

In this man page we describe how character classes are declared in the **stringi** package so that you may e.g. find their occurrences in your search activities or generate random code points with stri_rand_strings. Moreover, the **ICU** regex engine uses the same scheme for denoting character classes.

Details

All `stri_*_charclass` functions in **stringi** perform a single character (i.e. Unicode code point) search-based operations. Since stringi_0.2-1 you may obtain roughly the same results using stringi-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 either bounded by square brackets (such patterns follow a syntax similar to that employed by version 8 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". On the other hand, `[[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 backslashed (white spaces can be freely used for clarity, as `[a c d-f m]` means the same as `[acd-fm]`). **stringi** does not allow for including so-called multicharacter strings (see `UnicodeSet` API documentation). Also, empty string patterns are disallowed.

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

A malformed pattern always results in an error.

Set expressions at a glance (according to http://userguide.icu-project.org/strings/regexp):

Some examples:

`[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. (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}]` Logical AND or intersection – match the set of all Cyrillic letters.

`[\p{Letter}--\p{script=latin}]` Subtraction – match all non-Latin letters.

`[[a-z][A-Z][0-9]]` **or** `[a-zA-Z0-9]` Implicit Logical OR or Union of Sets – the examples match ASCII letters and digits. The two forms are equivalent.

`[:script=Greek:]` Alternate 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 properties' 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 perfectly documented. Please refer to the References section for some links.

Please note that some classes may seem to 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 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).

`Pe` a closing 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..FFFB, 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, cf. 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 ZWNBSP.

`CASE_SENSITIVE`

`POSIX_ALNUM`

`POSIX_BLANK`

`POSIX_GRAPH`

`POSIX_PRINT`

`POSIX_XDIGIT`

`CASED`

`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

Beware of using POSIX character classes, e.g. `[:punct:]`. ICU User Guide (see below) states that in general they are not well-defined, so 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 portable code (again, in POSIX-like regex engines).

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

References

The Unicode Character Database – Unicode Standard Annex #44, http://www.unicode.org/reports/tr44/

UnicodeSet – ICU User Guide, http://userguide.icu-project.org/strings/unicodeset

Properties – ICU User Guide, http://userguide.icu-project.org/strings/properties

C/POSIX Migration – ICU User Guide, http://userguide.icu-project.org/posix

Unicode Script Data, http://www.unicode.org/Public/UNIDATA/Scripts.txt

icu::Unicodeset Class Reference – ICU4C API Documentation, http://www.icu-project.org/apiref/icu4c/classicu_1_1UnicodeSet.html

See Also

Other search_charclass: stri_trim_both, stringi-search

Other stringi_general_topics: stringi-arguments, stringi-encoding, stringi-locale, stringi-package, stringi-search-boundaries, stringi-search-coll, stringi-search-fixed, stringi-search-regex, stringi-search

---

stringi-search-coll *Locale-Sensitive Text Searching in* **stringi**

Description

String searching facilities described in this very man page provide a way to locate a specific piece of text. Note that locale-sensitive searching, especially on a non-English text, is a much more complex process than it seems at the first glance.

Locale-Aware String Search Engine

All `stri_*_coll` functions in **stringi** utilize **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 often exhibit poor performance. These functions are not intended to be fast; they are made to give *correct* in natural language processing tasks.

References

ICU String Search Service – ICU User Guide, http://userguide.icu-project.org/collation/icu-string-search-service

L. Werner, *Efficient Text Searching in Java*, 1999, http://icu-project.org/docs/papers/efficient_text_searching_in_java.html

See Also

Other search_coll: stri_opts_collator, stringi-search

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries

Other stringi_general_topics: stringi-arguments, stringi-encoding, stringi-locale, stringi-package, stringi-search-boundaries, stringi-search-charclass, stringi-search-fixed, stringi-search-regex, stringi-search

---

stringi-search-fixed *Locale-Insensitive Fixed Pattern Matching in* **stringi**

Description

String searching facilities described in this very man page provide a way to locate a specific sequence of bytes in a string. Fixed pattern search engine's settings may be tuned up (for example to perform case-insensitive search), see the stri_opts_fixed function for more details.

Byte Compare

The Knuth-Morris-Pratt search algorithm, with worst time complexity of O(n+p) (`n == length(str)`, `p == length(pattern)`) is utilized (with some tweaks for very short search patterns). For natural language processing, however, this is not what you probably want. 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 stringi-search-coll.

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

See Also

Other search_fixed: stri_opts_fixed, stringi-search

Other stringi_general_topics: stringi-arguments, stringi-encoding, stringi-locale, stringi-package, stringi-search-boundaries, stringi-search-charclass, stringi-search-coll, stringi-search-regex, stringi-search

`stringi-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 in your hand 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), see the `stri_opts_regex` function for more details.

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.

`|` Alternation. A|B matches either A or B.

`*` Match 0 or more times. Match as many times as possible.

`+` Match 1 or more times. Match as many times as possible.

`?` Match zero or one times. Prefer one.

`{n}` Match exactly n times.

`{n,}` Match at least n times. Match as many times as possible.

`{n,m}` Match between n and m times. Match as many times as possible, but not more than m.

`*?` Match 0 or more times. Match as few times as possible.

`+?` Match 1 or more times. Match as few times as possible.

`??` Match zero or one times. Prefer zero.

`{n}?` Match exactly n times.

`{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 subexpression 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. First match of the parenthesized subexpression 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 position, 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 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.)

`(?<name>...)` Named capture group. The `<angle brackets>` are literal - they appear in the pattern.

`(?ismwx-ismwx:...)` Flag settings. Evaluate the parenthesized expression with the specified flags enabled or `-`disabled, see also `stri_opts_regex`.

`(?ismwx-ismwx)` 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 Metacharacters at a Glance

Here is a list of metacharacters 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. `stri_extract_all_words`.

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

`\cX` Match a control-`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, `\u001B`.

`\E` Terminates a `\Q` ... `\E` quoted sequence.

`\f` Match a FORM FEED, `\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, `\u0009`. [Since ICU 55]

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

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

`\n` Match a LINE FEED, `\u000A`.

`\N{UNICODE CHARACTER NAME}` Match the named character.

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

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

`\Q` Quotes all following characters until `\E`.

`\r` Match a CARRIAGE RETURN, `\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, `\u0009`.

`\uhhhh` 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 `\U0010ffff`.

`\w` Match a word character. Word characters are `[\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctua`

`\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 - &.

For information on how to define character classes in regexes, refer to stringi-search-charclass.

Regex Functions in stringi

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

If you would like to search for a fixed pattern, refer to stringi-search-coll or stringi-search-fixed. This allows to do a locale-aware text lookup, or a very fast exact-byte search, respectively.

References

Regular expressions – ICU User Guide, http://userguide.icu-project.org/strings/regexp

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

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

Unicode Regular Expressions – Regex tutorial, http://www.regular-expressions.info/unicode.html

See Also

Other search_regex: stri_opts_regex, stringi-search

Other stringi_general_topics: stringi-arguments, stringi-encoding, stringi-locale, stringi-package, stringi-search-boundaries, stringi-search-charclass, stringi-search-coll, stringi-search-fixed, stringi-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 appear in a specific lexicographic order.

Usage

```
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 as generated with stri_opts_collator, NULL for 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 allow to check whether there is any difference between them. These are locale-independent operations: for natural language text processing, in which 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, so e.g. stri_cmp_eq does not take BOMs into account while comparing strings.

On the other hand, 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.

What is more, 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<%.

Finally, 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 it up in **stringi**, refer to stri_opts_collator. Please 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 with comparison results of corresponding pairs of elements in e1 and e2: -1 if e1[...] < e2[...], 0 if they are canonically equivalent, and 1 if greater.

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

References

Collation - ICU User Guide, http://userguide.icu-project.org/collation

See Also

Other locale_sensitive: %s<%, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

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")

# 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 Matches*

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 with 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 defining search patterns; for more details refer to stringi-search
`opts_collator, opts_fixed, opts_regex`	a named list used to tune up a search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for default settings;

Details

Vectorized over `str` and `pattern`.

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; relying on one of those underlying functions will be faster.

Value

All the functions return an integer vector.

See Also

Other search_count: `stri_count_boundaries`, `stringi-search`

Examples

```
s <- "Lorem ipsum dolor sit amet, consectetur adipisicing elit."
stri_count(s, fixed="dolor")
stri_count(s, regex="\\p{L}+")

stri_count_fixed(s, " ")
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}") # whitespace - 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 baaab", c("b.*?b", "b.b"))
stri_count_regex(c("stringi", "123"), "^(s|1)")
```

`stri_count_boundaries` *Count the Number of Text Boundaries*

Description

This function determines the number of specific text boundaries (like character, word, line, or sentence boundaries) in a string.

Usage

```
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 as generated with `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 the text boundary analysis performed by **ICU**'s `BreakIterator`, see stringi-search-boundaries.

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 word boundaries, and all non-word characters (`UBRK_WORD_NONE` rule status) are ignored. This is function is equivalent to a call to `stri_count_boundaries(str, type="w`

Note that a `BreakIterator` of type `character` may be used to count the number of *Unicode characters* in a string. This may lead to different results than that returned by the `stri_length` function, which is designed to return the number of *Unicode code points*.

On the other hand, a `BreakIterator` of type `sentence` may be used to count the number of sentences in a piece of text.

Value

Both functions return an integer vector.

See Also

Other search_count: `stri_count`, `stringi-search`

Other locale_sensitive: `%s<%`, `stri_compare`, `stri_duplicated`, `stri_enc_detect2`, `stri_extract_all_boundaries`, `stri_locate_all_boundaries`, `stri_opts_collator`, `stri_order`, `stri_split_boundaries`, `stri_trans_tolower`, `stri_unique`, `stri_wrap`, `stringi-locale`, `stringi-search-boundaries`, `stringi-search-coll`

Other text_boundaries: `stri_extract_all_boundaries`, `stri_locate_all_boundaries`, `stri_opts_brkiter`, `stri_split_boundaries`, `stri_split_lines`, `stri_trans_tolower`, `stri_wrap`, `stringi-search-boundaries`, `stringi-search`

Examples

```
test <- "The\u00a0above-mentioned   features are very useful. Warm thanks to their developers."
stri_count_boundaries(test, type="word")
stri_count_boundaries(test, type="sentence")
stri_count_boundaries(test, type="character")
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 *Date and Time Arithmetic*

Description

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

Usage

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

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

Arguments

time	an object of class POSIXct or an object coercible to
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.

References

Calendar Classes - ICU User Guide, http://userguide.icu-project.org/datetime/calendar

See Also

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

```
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

This function constructs date-time objects from numeric representations.

Usage

```
stri_datetime_create(year, month, day, hour = 12L, minute = 0L,
  second = 0, lenient = FALSE, tz = NULL, locale = NULL)
```

Arguments

year	integer vector; 0 is 1BC, -1 is 2BC, etc.
month	integer vector; months are 1-based
day	integer vector
hour	integer vector
minute	integer vector
second	numeric vector; fractional seconds are allowed
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, hour, hour, minute, and second.

Value

Returns an object of class POSIXct.

See Also

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

```
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_fields *Get Values for Date and Time Fields*

Description

Calculates 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 or an object coercible to
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 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)

See Also

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")
stri_datetime_symbols(locale="@calendar=hebrew")$Month[
   stri_datetime_fields(stri_datetime_now(), locale="@calendar=hebrew")$Month
]
```

---

stri_datetime_format *Date and Time Formatting and Parsing*

Description

These functions convert a given date/time object to a character vector or conversely.

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 or an object coercible to
format	single string, 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_list

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
str	character vector
lenient	single logical value; should date/time parsing be lenient?
...	Further arguments to be passed from or to other methods.

Details

Vectorized over time or str.

By default, stri_datetime_format (unlike format.POSIXst) formats a date/time object using the current default time zone. This is for the sake of compatibility with the strftime function.

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, so 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 and time data from a calendar when formatting or used to generate data for a calendar when parsing. For example, y stands for the year. Characters may be used multiple times. For instance, if y is used for the year, yy might produce 99, whereas yyyy produces 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 produces 05. For some characters, the count specifies whether an abbreviated or full form should be used, but may have other choices, as given below.

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 letter 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 variable in the future (eg : being interpreted as the time separator and / as a date separator, and replaced by respective locale-sensitive characters in display).

Symbol	Meaning	Example(s)	Output
G	era designator	G, GG, or GGG	AD
		GGGG	Anno Domini
		GGGGG	A
y	year	yy	96
		y or yyyy	1996
u	extended year	u	4601
U	cyclic year name, as in Chinese lunar calendar	U	
r	related Gregorian year	r	1996
Q	quarter	Q or QQ	02
		QQQ	Q2
		QQQQ	2nd quarter
		QQQQQ	2
q	Stand Alone quarter	q or qq	02
		qqq	Q2
		qqqq	2nd quarter
		qqqqq	2
M	month in year	M or MM	09

		MMM	Sep
		MMMM	September
		MMMMM	S
L	Stand Alone month in year	L or LL	09
		LLL	Sep
		LLLL	September
		LLLLL	S
w	week of year	w or ww	27
W	week of month	W	2
d	day in month	d	2
		dd	02
D	day of year	D	189
F	day of week in month	F	2 (2nd Wed in July)
g	modified Julian day	g	2451334
E	day of week	E, EE, or EEE	Tue
		EEEE	Tuesday
		EEEEE	T
		EEEEEE	Tu
e	local day of week	e or ee	2
	example: if Monday is 1st day, Tuesday is 2nd)	eee	Tue
		eeee	Tuesday
		eeeee	T
		eeeeee	Tu
c	Stand Alone local day of week	c or cc	2
		ccc	Tue
		cccc	Tuesday
		ccccc	T
		cccccc	Tu
a	am/pm marker	a	pm
h	hour in am/pm (1~12)	h	7
		hh	07
H	hour in day (0~23)	H	0
		HH	00
k	hour in day (1~24)	k	24
		kk	24
K	hour in am/pm (0~11)	K	0
		KK	00
m	minute in hour	m	4
		mm	04
s	second in minute	s	5
		ss	05
S	fractional second - truncates (like other time fields)	S	2
	to the count of letters when formatting. Appends	SS	23
	zeros if more than 3 letters specified. Truncates at	SSS	235
	three significant digits when parsing.	SSSS	2350
A	milliseconds in day	A	61201235
z	Time Zone: specific non-location	z, zz, or zzz	PDT
		zzzz	Pacific Daylight Time

Z	Time Zone: ISO8601 basic hms? / RFC 822	Z, ZZ, or ZZZ	-0800
	Time Zone: long localized GMT (=OOOO)	ZZZZ	GMT-08:00
	Time Zone: ISO8601 extended hms? (=XXXXX)	ZZZZZ	-08:00, -07:52:58, Z
O	Time Zone: short localized GMT	O	GMT-8
	Time Zone: long localized GMT (=ZZZZ)	OOOO	GMT-08:00
v	Time Zone: generic non-location	v	PT
	(falls back first to VVVV)	vvvv	Pacific Time or Los Angeles Time
V	Time Zone: short time zone ID	V	uslax
	Time Zone: long time zone ID	VV	America/Los_Angeles
	Time Zone: time zone exemplar city	VVV	Los Angeles
	Time Zone: generic location (falls back to OOOO)	VVVV	Los Angeles Time
X	Time Zone: ISO8601 basic hm?, with Z for 0	X	-08, +0530, Z
	Time Zone: ISO8601 basic hm, with Z	XX	-0800, Z
	Time Zone: ISO8601 extended hm, with Z	XXX	-08:00, Z
	Time Zone: ISO8601 basic hms?, with Z	XXXX	-0800, -075258, Z
	Time Zone: ISO8601 extended hms?, with Z	XXXXX	-08:00, -07:52:58, Z
x	Time Zone: ISO8601 basic hm?, without Z for 0	x	-08, +0530
	Time Zone: ISO8601 basic hm, without Z	xx	-0800
	Time Zone: ISO8601 extended hm, without Z	xxx	-08:00
	Time Zone: ISO8601 basic hms?, without Z	xxxx	-0800, -075258
	Time Zone: ISO8601 extended hms?, without Z	xxxxx	-08:00, -07:52:58
’	escape for text	’	(nothing)
’ ’	two single quotes produce one	’ ’	’

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 they are not enclosed within single quotes. The single quote is used to “escape” letters. Two single quotes in a row, inside or outside a quoted sequence, represent a “real” single quote.

Here are some examples:

Exemplary Pattern	**Result**
yyyy.MM.dd ’at’ HH:mm:ss zzz	2015.12.31 at 23:59:59 GMT+1
EEE, MMM d, ”yy	czw., gru 31, ’15
h:mm a	11:59 PM
hh ’o”clock’ a, zzzz	11 o’clock PM, GMT+01:00
K:mm a, z	11:59 PM, GMT+1
yyyyy.MMMM.dd GGG hh:mm aaa	2015.grudnia.31 n.e. 11:59 PM
uuuu-MM-dd’T’HH:mm:ssZ	2015-12-31T23:59:59+0100 (the ISO 8601 guideline)

Value

`stri_datetime_format` returns a character vector.

`stri_datetime_parse` returns an object of class `POSIXct`.

References

Formatting Dates and Times - ICU User Guide, http://userguide.icu-project.org/formatparse/datetime

See Also

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

```
stri_datetime_parse(c("2015-02-28", "2015-02-29"), "yyyy-MM-dd")
stri_datetime_parse(c("2015-02-28", "2015-02-29"), "yyyy-MM-dd", lenient=TRUE)
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

A function to convert strptime/strftime-style format strings to **ICU** format strings that may be used in stri_datetime_parse and stri_datetime_format functions.

Usage

```
stri_datetime_fstr(x)
```

Arguments

x	character vector consisting of date/time format strings

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

Value

Returns a character vector.

See Also

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

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

`stri_datetime_now` *Get Current Date and Time*

Description

Returns current date and time.

Usage

```
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`.

See Also

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

`context` stands for a selector for date formatting context and `width` - for date formatting width.

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.

References

Calendar - ICU User Guide, http://userguide.icu-project.org/datetime/calendar

DateFormatSymbols class – ICU API Documentation, http://icu-project.org/apiref/icu4c/classicu_1_1DateFormatSymbols.html

Formatting Dates and Times – ICU User Guide, http://userguide.icu-project.org/formatparse/datetime

See Also

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

```
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=japanese")

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")
stri_datetime_symbols(width="abbreviated")
stri_datetime_symbols(width="narrow")
```

stri_detect *Detect a Pattern Match*

Description

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

Usage

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

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

stri_detect_charclass(str, pattern, negate = FALSE)

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

stri_detect_regex(str, pattern, negate = FALSE, ..., opts_regex = NULL)
```

Arguments

`str`	character vector with 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 defining search patterns; for more details refer to stringi-search
`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 a search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for default settings;

Details

Vectorized over `str` and `pattern`.

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. Relying on these underlying functions will make your code run slightly faster.

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

Value

Each function returns a logical vector.

See Also

Other search_detect: stri_startswith, stringi-search

Examples

```
stri_detect_fixed(c("stringi R", "REXAMINE", "123"), c('i', 'R', '0'))
stri_detect_fixed(c("stringi R", "REXAMINE", "123"), 'R')

stri_detect_charclass(c("stRRRingi","REXAMINE", "123"),
   c("\\p{Ll}", "\\p{Lu}", "\\p{Zs}"))

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

---

stri_dup *Duplicate Strings*

Description

Duplicates each string `times` times and concatenates the results.

Usage

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

Arguments

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

Details

Vectorized over `str` and `times`.

Value

Returns a character vector of the same length as `str`.

See Also

Other join: stri_flatten, stri_join_list, stri_join

Examples

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

`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

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

stri_duplicated_any(str, fromLast = FALSE, ..., opts_collator = NULL)
```

Arguments

`str`	a character vector
`fromLast`	a single logical value; indicating whether duplication should be considered from the reverse side
`...`	additional settings for `opts_collator`
`opts_collator`	a named list with **ICU** Collator's options as generated with `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 bytewise 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 counterpart.

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.

References

Collation - ICU User Guide, http://userguide.icu-project.org/collation

See Also

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_enc_detect2, stri_extract_all_boundar
stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries,
stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries,
stringi-search-coll

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_enc_detect2, stri_extract_all_boundar
stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries,
stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries,
stringi-search-coll

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), fromLast=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 a character vector between encodings.

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 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 shall be returned rather than a character vector

Details

stri_conv is an alias for stri_encode.

These two functions aim to replace R's iconv. It is not only faster, but also works in the same manner on all platforms.

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

If str is a character vector and from is either missing, "", or NULL, then the declared encodings are used (see stri_enc_mark) – in such a case bytes-declared strings are disallowed. Otherwise, the internal encoding declarations are ignored and a converter selected with from is used.

On the other hand, for str being a raw vector or a list of raw vectors, we assume that the input encoding is the current default encoding as given by stri_enc_get.

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

Note that problems may occur if to indicates e.g UTF-16 or UTF-32, as the output strings may have embedded NULs. In such cases use 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 BOMs).

Note that stri_encode(as.raw(data), "encodingname") is a wise substitute for rawToChar.

In the current version of **stringi**, if an incorrect code point is found on input, it is replaced by the default (for that target encoding) substitute character and a warning is generated.

Value

If to_raw is FALSE, then a character vector with encoded strings (and sensible encoding marks) is returned. Otherwise, a list of raw vectors is produced.

References

Conversion – ICU User Guide, http://userguide.icu-project.org/conversion

Converters – ICU User Guide, http://userguide.icu-project.org/conversion/converters (technical details)

See Also

Other encoding_conversion: stri_enc_fromutf32, stri_enc_toascii, stri_enc_tonative, stri_enc_toutf32, stri_enc_toutf8, stringi-encoding

`stri_enc_detect` *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

```
stri_enc_detect(str, filter_angle_brackets = FALSE)
```

Arguments

`str`	character vector, a raw vector, or a list of `raw` vectors
`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 `str` and `filter_angle_brackets`.

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's mostly in a single language. However, because the detection only looks at a limited amount of the input data, some of the returned charsets 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 `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:

Character_Set	Languages
UTF-8	–
UTF-16BE	–
UTF-16LE	–
UTF-32BE	–
UTF-32LE	–

Shift_JIS	Japanese
ISO-2022-JP	Japanese
ISO-2022-CN	Simplified Chinese
ISO-2022-KR	Korean
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

If you have some initial guess at language and encoding, try with `stri_enc_detect2`.

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 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.

References

Character Set Detection – ICU User Guide, http://userguide.icu-project.org/conversion/detection

See Also

Other encoding_detection: stri_enc_detect2, stri_enc_isascii, stri_enc_isutf16be, stri_enc_isutf8, stringi-encoding

Examples

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

## End(Not run)
```

---

stri_enc_detect2 *Detect Locale-Sensitive Character Encoding*

Description

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

Usage

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

Arguments

str	character vector, a raw vector, or a list of raw vectors
locale	NULL or "" for default locale, NA for just checking the UTF-* family, 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 slightly bases on **ICU**'s i18n/csrucode.cpp, but we do it in our own way, however) 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's in a single language.

If you have no initial guess on language and encoding, try with stri_enc_detect (uses **ICU** facilities). However, it turns out that (empirically) stri_enc_detect2 works better than the **ICU**-based one if UTF-* text is provided. Try it yourself.

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 (iff 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.

See Also

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_extract_all_boundari
stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries,
stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries,
stringi-search-coll

Other encoding_detection: stri_enc_detect, stri_enc_isascii, stri_enc_isutf16be, stri_enc_isutf8,
stringi-encoding

---

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)
```

Arguments

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 32bit encoding in which each Unicode code point corresponds to exactly one integer value.

This function roughly acts like 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 incorrect code point is given, a warning is generated and a 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_.

See Also

Other encoding_conversion: stri_enc_toascii, stri_enc_tonative, stri_enc_toutf32, stri_enc_toutf8, stri_encode, stringi-encoding

---

stri_enc_info *Query a Character Encoding*

Description

Gets basic information on a character encoding.

Usage

```
stri_enc_info(enc = NULL)
```

Arguments

enc	NULL or "" for default encoding, or a single string with encoding name

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 an UChar (in UTF-16, this is not the same as UChar32)
- CharSize.max – maximal number of bytes used to represent an UChar (in UTF-16, this is not the same as UChar32, i.e. does not reflect the maximal code point representation size)

See Also

Other encoding_management: stri_enc_list, stri_enc_mark, stri_enc_set, stringi-encoding

---

stri_enc_isascii *Check If a Data Stream Is Possibly in ASCII*

Description

The function checks whether all bytes in a string are in the set 1,2,...,127.

Usage

```
stri_enc_isascii(str)
```

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. Its i-th element indicates whether the i-th string corresponds to a valid ASCII byte sequence.

See Also

Other encoding_detection: stri_enc_detect2, stri_enc_detect, stri_enc_isutf16be, stri_enc_isutf8, stringi-encoding

Examples

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

stri_enc_isutf16be *Check If a Data Stream Is Possibly in UTF16 or UTF32*

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)
```

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 and stringi-encoding). Anyway, most often, you will provide input data as raw vectors here.

Negative answer means that a string is surely not in valid UTF-16 or UTF-32. Positive result does not mean that we should be absolutely sure.

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

Value

Returns a logical vector.

See Also

Other encoding_detection: stri_enc_detect2, stri_enc_detect, stri_enc_isascii, stri_enc_isutf8, stringi-encoding

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

Negative answer means that a string is surely not valid UTF-8. Positive result does not mean that we should be absolutely sure. E.g. `(c4,85)` properly represents ("Polish 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, have ASCII as their subsets (note that stri_enc_isascii => stri_enc_isutf8).

However, the longer the sequence, the bigger 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.

See Also

Other encoding_detection: stri_enc_detect2, stri_enc_detect, stri_enc_isascii, stri_enc_isutf16be, stringi-encoding

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 = FALSE)
```

Arguments

simplify	single logical value; return a character vector or a list of character vectors?

Details

Please note that apart from given encoding identifiers and their aliases, some other specifiers will be available in your conversion tasks. This is due to the fact that **ICU** tries to normalize converter names. E.g. "UTF8" is also valid, see stringi-encoding for more information.

Value

If simplify is FALSE (the default), 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, then the resulting list is coerced to a character vector and sorted, and returned with removed duplicated entries.

See Also

Other encoding_management: stri_enc_info, stri_enc_mark, stri_enc_set, stringi-encoding

stri_enc_mark *Get Declared Encodings of Each String*

Description

Gets 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, via the C API we may check 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 which is an ASCII superset) 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 when inputting data via 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 Encoding, possible encodings are: ASCII, latin1, bytes, native, and UTF-8. Additionally, missing values are handled properly.

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

See Also

Other encoding_management: stri_enc_info, stri_enc_list, stri_enc_set, stringi-encoding

---

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 currently used default encoding.

Usage

```
stri_enc_set(enc)

stri_enc_get()
```

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.

Value

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

stri_enc_get returns a string with current default character encoding.

See Also

Other encoding_management: stri_enc_info, stri_enc_list, stri_enc_mark, stringi-encoding

---

stri_enc_toascii *Convert To ASCII*

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)
```

Arguments

str	a character vector to be converted

Details

All code points greater than 127 are replaced with ASCII SUBSTITUTE CHARACTER (0x1A). R encoding declarations are always used to determine which encoding is assumed for each input, see stri_enc_mark. In incorrect byte sequences are found in UTF-8 byte streams, a warning is generated.

A bytes-marked string is assumed to be represented by a 8-bit encoding such that it has ASCII as its subset (a common assumption in R itself).

Note that the SUBSTITUTE CHARACTER (\x1a == \032) may be interpreted as ASCII missing value for single characters.

Value

Returns a character vector.

See Also

Other encoding_conversion: stri_enc_fromutf32, stri_enc_tonative, stri_enc_toutf32, stri_enc_toutf8, stri_encode, stringi-encoding

---

`stri_enc_tonative` *Convert Strings To Native Encoding*

Description

Converts character strings with declared encodings to 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)`. 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.

See Also

Other encoding_conversion: stri_enc_fromutf32, stri_enc_toascii, stri_enc_toutf32, stri_enc_toutf8, stri_encode, stringi-encoding

stri_enc_toutf32 *Convert Strings To UTF-32*

Description

UTF-32 is a 32bit encoding in which each Unicode code point corresponds to exactly one integer value. This function converts a character vector to a list of integer vectors so that e.g. individual code points may easily be accessed, changed, etc.

Usage

```
stri_enc_toutf32(str)
```

Arguments

str	a character vector (or an object coercible to such a vector) to be converted

Details

See stri_enc_fromutf32 for a dual operation.

This function is roughly equivalent to a vectorized call to utf8ToInt(enc2utf8(str)). If you want a list of raw vector on output, use stri_encode.

Unlike utf8ToInt, if improper UTF-8 byte sequences are detected, a corresponding element is set to NULL and a warning is given, see also stri_enc_toutf8 for a method to deal with such cases.

Value

Returns a list of integer vectors. Missing values are converted to NULLs.

See Also

Other encoding_conversion: stri_enc_fromutf32, stri_enc_toascii, stri_enc_tonative, stri_enc_toutf8, stri_encode, stringi-encoding

stri_enc_toutf8 *Convert Strings To UTF-8*

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 BOM will be silently removed from the output string.

If 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 characters. Here, a bytes-marked string is assumed to be encoded by an 8-bit encoding such that it has ASCII as its subset.

What is more, in both cases setting validate to TRUE or NA validates the resulting UTF-8 byte stream. If validate=TRUE, then in case of any incorrect byte sequences, they will be replaced with REPLACEMENT CHARACTER. This option may be used in a (very rare in practice) case in which 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.

See Also

Other encoding_conversion: stri_enc_fromutf32, stri_enc_toascii, stri_enc_tonative, stri_enc_toutf32, stri_encode, stringi-encoding

---

stri_escape_unicode *Escape Unicode Code Points*

Description

Escapes all Unicode (not ASCII-printable) code points.

Usage

```
stri_escape_unicode(str)
```

Arguments

str	character vector

Details

For non-printable and certain special (well-known, see also 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, any input string is converted to Unicode before executing the escape process.

Value

Returns a character vector.

See Also

Other escape: stri_unescape_unicode

Examples

```
stri_escape_unicode("a\u0105!")
```

---

stri_extract_all *Extract Occurrences of a Pattern*

Description

These functions extract all substrings matching a given pattern.

stri_extract_all_* extracts all the matches. On the other hand, stri_extract_first_* and stri_extract_last_* provide 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 with 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 defining search patterns; for more details refer to stringi-search
`merge`	single logical value; should consecutive matches 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 used to tune up a search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for default settings;

Details

Vectorized over `str` and `pattern`.

If you would like to extract regex capture groups individually, check out stri_match.

stri_extract, stri_extract_all, stri_extract_first, and stri_extract_last are convenience functions. They just call stri_extract_*_*, depending on the arguments used. Relying on one of those underlying functions will make your code run slightly faster.

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 separate 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, 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.

stri_extract_first* and stri_extract_last*, on the other hand, return a character vector. A NA element indicates no match.

See Also

Other search_extract: stri_extract_all_boundaries, stri_match_all, stringi-search

Examples

```
stri_extract_all('XaaaaX', 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 whitespaces

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

stri_list2matrix(stri_extract_all_regex('XaaaaX', c('\\p{Ll}', '\\p{Ll}+')))
stri_extract_all_regex('XaaaaX', c('\\p{Ll}', '\\p{Ll}+'), simplify=TRUE)
stri_extract_all_regex('XaaaaX', 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)
```

---

stri_extract_all_boundaries

Extract Text Between Text Boundaries

Description

These functions extract text between specific 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 as generated with 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 the 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 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 equal to TRUE and NA, respectively.

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

See Also

Other search_extract: stri_extract_all, stri_match_all, stringi-search

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

Other text_boundaries: stri_count_boundaries, stri_locate_all_boundaries, stri_opts_brkiter, stri_split_boundaries, stri_split_lines, stri_trans_tolower, stri_wrap, stringi-search-boundaries, stringi-search

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

```
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?
omit_empty	single logical value; should missing values 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 first string will be used.

Value

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

See Also

Other join: `stri_dup`, `stri_join_list`, `stri_join`

Examples

```
stri_flatten(LETTERS)
stri_flatten(LETTERS, collapse=",")
stri_flatten(c('abc', '123', '\u0105\u0104'))
stri_flatten(stri_dup(letters[1:6],1:3))
stri_flatten(c(NA, "", "A", "", "B", NA, "C"), collapse=",", na_empty=TRUE, omit_empty=TRUE)
```

---

stri_info *Query Default Settings for* **stringi**

Description

Presents current default settings used by the **ICU** library.

Usage

```
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=TRUE`, then a single string containing information on default character encoding, locale, and Unicode as well as **ICU** version is returned.

Otherwise, you 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` – always `c("UTF-8", "UTF-16")`;
- `Charset.native` – information on default encoding, as returned by `stri_enc_info`;
- `ICU.system` – logical; indicates whether system **ICU** libs are used (`TRUE`) or if **ICU** was built together with **stringi**.

`stri_isempty` *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, as opposed to the built-in `nzchar` function.

Value

Returns a logical vector of the same length as `str`.

See Also

Other length: `stri_length`, `stri_numbytes`, `stri_width`

Examples

```
stri_isempty(letters[1:3])
stri_isempty(c(',', '', 'abc', '123', '\u0105\u0104'))
stri_isempty(character(1))
```

stri_join *Concatenate Character Vectors*

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) which 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 on input 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=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 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.

See Also

Other join: stri_dup, stri_flatten, stri_join_list

Examples

```
stri_join(1:13, letters)
stri_join(1:13, letters, sep='!')
stri_join(1:13, letters, collapse='?')
stri_join(1:13, letters, sep='!', collapse='?')
stri_join(c('abc', '123', '\u0105\u0104'),'###', 1:5, sep='...')
stri_join(c('abc', '123', '\u0105\u0104'),'###', 1:5, sep='...', collapse='?')

do.call(stri_c, list(c("a", "b", "c"), c("1", "2"), sep='!'))
do.call(stri_c, list(c("a", "b", "c"), c("1", "2"), sep='!', collapse='$'))
```

---

stri_join_list *Concatenate Strings in a List*

Description

These functions concatenate 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.

Value

Returns a character vector.

See Also

Other join: stri_dup, stri_flatten, stri_join

Examples

```
stri_join_list(stri_extract_all_words(c("Lorem ipsum dolor sit amet.",
"You're gonna get away with this.")), sep=", ")

stri_join_list(stri_extract_all_words(c("Lorem ipsum dolor sit amet.",
"You're gonna get away with this.")), sep=", ", collapse=". ")

stri_join_list(stri_extract_all_regex(c("R is OK.", "123 456", "Hey!"), "\\p{L}+"), " ")

stri_join_list(stri_extract_all_regex(c("R is OK.", "123 456", "Hey!"),
"\\p{L}+", omit_no_match=TRUE), " ", " -- ")
```

---

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 screen.

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, as opposed to the built-in nchar function. For 'byte' encodings we get, as usual, an error.

Value

Returns an integer vector of the same length as str.

See Also

Other length: stri_isempty, stri_numbytes, 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)
```

Arguments

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

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

Always returns a character matrix.

See Also

Other utils: stri_na2empty, stri_remove_empty

Examples

```
simplify2array(list(c("a", "b"), c("c", "d"), c("e", "f")))
stri_list2matrix(list(c("a", "b"), c("c", "d"), c("e", "f")))
stri_list2matrix(list(c("a", "b"), c("c", "d"), c("e", "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

```
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 nothing complicated. In many cases it is similar to a call to as.list(stri_split_fixed(locale, "_", with locale case mapped. It may be used, however, to get insight on how ICU understands a provided locale identifier.

Value

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

See Also

Other locale_management: stri_locale_list, stri_locale_set, stringi-locale

Examples

```
stri_locale_info("pl_PL")
stri_locale_info("Pl_pL") # the same result
```

---

stri_locale_list *List Available Locales*

Description

Creates a character vector with all known locale identifies.

Usage

```
stri_locale_list()
```

Details

Note that not all services may be available for all locales. Queries for locale-specific services are 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**.

See Also

Other locale_management: `stri_locale_info`, `stri_locale_set`, `stringi-locale`

---

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

Description

`stri_locale_set` changes default locale for all functions in the **stringi** package, i.e. establishes the meaning of the "`NULL` locale" argument of locale-sensitive functions. On the other hand, `stri_locale_get` gets current default locale.

Usage

```
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 default locale.

stri_locale_get 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".

See Also

Other locale_management: stri_locale_info, stri_locale_list, stringi-locale

Examples

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

## End(Not run)
```

---

stri_locate_all *Locate Occurrences of a Pattern*

Description

These functions may be used e.g. to find the indices (positions), at which a given pattern is matched. stri_locate_all_* locate all the matches. On the other hand, stri_locate_first_* and stri_locate_last_* give the first or 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)

stri_locate_first_charclass(str, pattern)

stri_locate_last_charclass(str, pattern)

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

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

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

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

stri_locate_first_regex(str, pattern, ..., opts_regex = NULL)

stri_locate_last_regex(str, pattern, ..., opts_regex = NULL)

stri_locate_all_fixed(str, pattern, omit_no_match = FALSE, ...,
  opts_fixed = NULL)

stri_locate_first_fixed(str, pattern, ..., opts_fixed = NULL)

stri_locate_last_fixed(str, pattern, ..., opts_fixed = NULL)
```

Arguments

`str`	character vector with 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 defining search patterns; for more details refer to stringi-search
`merge`	single logical value; indicates whether consecutive sequences of indices in the resulting matrix shall be merged; `stri_locate_all_charclass` only
`omit_no_match`	single logical value; if `FALSE`, then 2 missing values will indicate that there was no match; `stri_locate_all_*` only
`opts_collator, opts_fixed, opts_regex`	a named list used to tune up a search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for default settings;

Details

Vectorized over `str` and `pattern`.

The matched string(s) may be extracted by calling the `stri_sub` function. 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 arguments used. Unless you are a very lazy person, please call the underlying functions directly for better performance.

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 matches, and the second column gives the end positions. Moreover, you may get two `NA`s in one row for no match (if `omit_no_match` is `FALSE`) or `NA` arguments.

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

For `stri_locate_*_regex`, if the match is of length 0, `end` will be one character less than `start`.

See Also

Other search_locate: `stri_locate_all_boundaries`, `stringi-search`

Other indexing: `stri_locate_all_boundaries`, `stri_sub`

Examples

```
stri_locate_all('XaaaaX',
   regex=c('\\p{Ll}', '\\p{Ll}+', '\\p{Ll}{2,3}', '\\p{Ll}{2,3}?'))
stri_locate_all('Bartolini', fixed='i')
stri_locate_all('a b c', charclass='\\p{Zs}') # all white spaces

stri_locate_all_charclass(c('AbcdeFgHijK', 'abc', 'ABC'), '\\p{Ll}')
stri_locate_all_charclass(c('AbcdeFgHijK', 'abc', 'ABC'), '\\p{Ll}', merge=FALSE)
stri_locate_first_charclass('AaBbCc', '\\p{Ll}')
stri_locate_last_charclass('AaBbCc', '\\p{Ll}')

stri_locate_all_coll(c('AaaaaaaA', 'AAAA'), 'a')
stri_locate_first_coll(c('Yy\u00FD', 'AAA'), 'y', strength=2, locale="sk_SK")
stri_locate_last_coll(c('Yy\u00FD', 'AAA'), 'y', strength=1, locale="sk_SK")

pat <- stri_paste("\u0635\u0644\u0649 \u0627\u0644\u0644\u0647 ",
                  "\u0639\u0644\u064a\u0647 \u0648\u0633\u0644\u0645XYZ")
stri_locate_last_coll("\ufdfa\ufdfa\ufdfaXYZ", pat, strength = 1)

stri_locate_all_fixed(c('AaaaaaaA', 'AAAA'), 'a')
stri_locate_all_fixed(c('AaaaaaaA', 'AAAA'), 'a', case_insensitive=TRUE, overlap=TRUE)
stri_locate_first_fixed(c('AaaaaaaA', 'aaa', 'AAA'), 'a')
stri_locate_last_fixed(c('AaaaaaaA', 'aaa', 'AAA'), 'a')
```

```
#first row is 1-2 like in locate_first
stri_locate_all_fixed('bbbbb', 'bb')
stri_locate_first_fixed('bbbbb', 'bb')

# but last row is 3-4, unlike in locate_last,
# keep this in mind [overlapping pattern match OK]!
stri_locate_last_fixed('bbbbb', 'bb')

stri_locate_all_regex('XaaaaX',
   c('\\p{Ll}', '\\p{Ll}+', '\\p{Ll}{2,3}', '\\p{Ll}{2,3}?'))
stri_locate_first_regex('XaaaaX',
   c('\\p{Ll}', '\\p{Ll}+', '\\p{Ll}{2,3}', '\\p{Ll}{2,3}?'))
stri_locate_last_regex('XaaaaX',
   c('\\p{Ll}', '\\p{Ll}+', '\\p{Ll}{2,3}', '\\p{Ll}{2,3}?'))

# Use regex positive-lookahead to locate overlapping pattern matches:
stri_locate_all_regex("ACAGAGACTTTAGATAGAGAAGA", "(?=AGA)")
# note that start > end here (match of 0 length)
```

`stri_locate_all_boundaries`

Locate Specific Text Boundaries

Description

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

Usage

```
stri_locate_all_boundaries(str, omit_no_match = FALSE, ...,
  opts_brkiter = NULL)

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

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

stri_locate_all_words(str, omit_no_match = FALSE, locale = NULL)

stri_locate_last_words(str, locale = NULL)

stri_locate_first_words(str, locale = NULL)
```

Arguments

str	character vector or an object coercible to
omit_no_match	single logical value; if FALSE, then 2 missing values will indicate that there are no text boundaries
...	additional settings for opts_brkiter
opts_brkiter	a named list with **ICU** BreakIterator's settings as generated with 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 the text boundary analysis performed by **ICU**'s BreakIterator, see stringi-search-boundaries.

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

Value

For stri_locate_all_*, a list of length(str) integer matrices is returned. The first column gives the start positions of substrings between located boundaries, and the second column gives the end positions. The indices are code point-based, thus they may be passed e.g. to the stri_sub function. Moreover, you may get two NAs in one row for no match (if omit_no_match is FALSE) or NA arguments.

stri_locate_first_* and stri_locate_last_*, on the other hand, 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.

See Also

Other search_locate: stri_locate_all, stringi-search

Other indexing: stri_locate_all, stri_sub

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

Other text_boundaries: stri_count_boundaries, stri_extract_all_boundaries, stri_opts_brkiter, stri_split_boundaries, stri_split_lines, stri_trans_tolower, stri_wrap, stringi-search-boundaries, stringi-search

Examples

```
test <- "The\u00a0above-mentioned   features are very useful. Warm thanks to their developers."
stri_locate_all_boundaries(test, type="line")
stri_locate_all_boundaries(test, type="word")
stri_locate_all_boundaries(test, type="sentence")
stri_locate_all_boundaries(test, type="character")
stri_locate_all_words(test)

stri_extract_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
```

---

stri_match_all *Extract Regex Pattern Matches, Together with Capture Groups*

Description

These functions extract substrings of `str` that match a given regex `pattern`. Additionally, they extract matches to every *capture group*, i.e. to all the subpatterns given in round parentheses.

Usage

```
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(str, pattern, omit_no_match = FALSE,
  cg_missing = NA_character_, ..., opts_regex = NULL)

stri_match_first_regex(str, pattern, cg_missing = NA_character_, ...,
  opts_regex = NULL)

stri_match_last_regex(str, pattern, cg_missing = NA_character_, ...,
  opts_regex = NULL)
```

Arguments

`str`	character vector with strings to search in
`...`	supplementary arguments passed to the underlying functions, including additional settings for `opts_regex`
`mode`	single string; one of: `"first"` (the default), `"all"`, `"last"`
`pattern, regex`	character vector defining regex patterns to search for; for more details refer to stringi-search-regex

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 as generated with stri_opts_regex; NULL for default settings;

Details

Vectorized over str and pattern.

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

Please note: **ICU** regex engine currently does not support named capture groups.

stri_match, stri_match_all, stri_match_first, and stri_match_last are convenience functions. They just call stri_match_*_regex – they have been provided for consistency with other string searching functions' wrappers, cf. e.g. stri_extract.

Value

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

For stri_match_first* and stri_match_last*, on the other hand, a character matrix is returned. Here the search results are provided as separate rows.

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.

See Also

Other search_extract: stri_extract_all_boundaries, stri_extract_all, stringi-search

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+)")
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

Usage

```
stri_na2empty(x)
```

Arguments

x	a character vector

Value

Always returns a character vector.

See Also

Other utils: stri_list2matrix, stri_remove_empty

Examples

```
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 computer's memory.

Usage

```
stri_numbytes(str)
```

Arguments

str	character vector or an object coercible to

Details

This is often not the function you would normally use in your string processing activities. See rather stri_length.

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, as opposed to the built-in nchar(str, "bytes") function call.

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

The returned values does not of course 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.

See Also

Other length: stri_isempty, stri_length, stri_width

Examples

```
stri_numbytes(letters)
stri_numbytes(c('abc', '123', '\u0105\u0104'))

## Not run:
# this used to fail on Windows, as there was no native support for 4-bytes
# Unicode characters; see, however, stri_escape_unicode():
stri_numbytes('\U7fffffff') # compare stri_length('\U7fffffff')

## End(Not run)
```

---

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

```
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, ...)
```

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 at which 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
...	any other arguments to this function are purposely ignored

Details

The skip_* family of settings may be used to prevent performing any special actions on particular types 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.

References

ubrk.h *File Reference* – ICU4C API Documentation, http://icu-project.org/apiref/icu4c/ubrk_8h.html

Boundary Analysis – ICU User Guide, http://userguide.icu-project.org/boundaryanalysis

See Also

Other text_boundaries: stri_count_boundaries, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_split_boundaries, stri_split_lines, stri_trans_tolower, stri_wrap, stringi-search-boundaries, stringi-search

---

`stri_opts_collator` *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

```
stri_opts_collator(locale = NULL, strength = 3L,
  alternate_shifted = FALSE, french = FALSE, uppercase_first = NA,
  case_level = FALSE, normalization = FALSE, 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 most strict 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
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
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'
...	any other arguments to this function are purposely ignored

Details

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

A note on collation `strength`: generally, `strength` set to 4 is the least permissive. Set to 2 to ignore case differences. Set to 1 to also ignore diacritical differences.

The strings are Unicode-normalized before the comparison.

Value

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

References

Collation – ICU User Guide, http://userguide.icu-project.org/collation

ICU Collation Service Architecture – ICU User Guide, http://userguide.icu-project.org/collation/architecture

`icu::Collator` *Class Reference* – ICU4C API Documentation, http://www.icu-project.org/apiref/icu4c/classicu_1_1Collator.html

See Also

Other locale_sensitive: `%s<%`, `stri_compare`, `stri_count_boundaries`, `stri_duplicated`, `stri_enc_detect2`, `stri_extract_all_boundaries`, `stri_locate_all_boundaries`, `stri_order`, `stri_split_boundaries`, `stri_trans_tolower`, `stri_unique`, `stri_wrap`, `stringi-locale`, `stringi-search-boundaries`, `stringi-search-coll`

Other search_coll: `stringi-search-coll`, `stringi-search`

Examples

```
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 `stri_*_fixed` functions' behavior, see stringi-search-fixed.

Usage

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

Arguments

case_insensitive	logical; enable simple case insensitive matching
overlap	logical; enable overlapping matches detection in certain functions
...	any other arguments to this function are purposely ignored

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 take into account the string context and the language and can map to a result string with a different length as appropriate, see stringi-search-coll.

Searching for overlapping pattern matches works in case of the stri_extract_all_fixed, stri_locate_all_fixed, and stri_count_fixed functions.

Value

Returns a named list object.

References

C/POSIX Migration – ICU User Guide, http://userguide.icu-project.org/posix#case_mappings

See Also

Other search_fixed: stringi-search-fixed, stringi-search

Examples

```
stri_detect_fixed("ala", "ALA") # case-sensitive by default
stri_detect_fixed("ala", "ALA", opts_fixed=stri_opts_fixed(case_insensitive=TRUE))
stri_detect_fixed("ala", "ALA", case_insensitive=TRUE) # equivalent
```

---

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

```
stri_opts_regex(case_insensitive, comments, dotall, literal, multiline,
  unix_lines, uword, error_on_unknown_escapes, ...)
```

Arguments

`case_insensitive`
: logical; enable case insensitive matching [regex flag `(?i)`]

`comments`
: logical; allow 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)`]

`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)`]

`unix_lines`
: logical; Unix-only line endings. When this mode is enabled, only `U+000a` is recognized as a line ending by '`.`', '`$`', and '`^`'.

`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 http://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

`...`
: any other arguments to this function are purposely ignored

Details

Note that some regex settings may be changed using ICU regex flags inside regexes. For example, `"(?i)pattern"` does 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.

References

`enum URegexpFlag`: *Constants for Regular Expression Match Modes* – ICU4C API Documentation, http://www.icu-project.org/apiref/icu4c/uregex_8h.html

Regular Expressions – ICU User Guide, http://userguide.icu-project.org/strings/regexp

See Also

Other search_regex: `stringi-search-regex`, `stringi-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 and Sorting*

Description

stri_order determines a permutation which rearranges strings into an ascending or descending order. stri_sort sorts the vector according to a lexicographic order.

Usage

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

stri_sort(str, decreasing = FALSE, na_last = NA, ...,
  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 as generated with 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.

These functions use a stable sort algorithm (**STL**'s stable_sort), which performs up to $N * log^2(N)$ element comparisons, where N is the length of str.

Interestingly, our benchmarks indicate that stri_order is most often faster that R's order.

Value

For stri_order, an integer vector that gives the sort order is returned.

For stri_order, you get a sorted version of str, i.e. a character vector.

References

Collation - ICU User Guide, http://userguide.icu-project.org/collation

See Also

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

Examples

```
stri_sort(c("hladny", "chladny"), locale="pl_PL")

stri_sort(c("hladny", "chladny"), locale="sk_SK")
```

---

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

Description

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

Usage

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

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

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

stri_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, 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.

`stri_pad` is a convenience function, which dispatches control to `stri_pad_*`. Relying on one of the underlying functions will make your code run slightly faster.

Note that Unicode code points may have various widths when printed on the console and that the function takes that by default into account. By changing the state of the `use_length` argument, this function starts to act 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

Returns a character vector.

Examples

```
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("\ud6c8\ubbfc\uc815\uc74c", # takes width into account
   stri_trans_nfkd("\ud6c8\ubbfc\uc815\uc74c"), "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

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

Arguments

`nparagraphs`	single integer, number of paragraphs to generate
`start_lipsum`	single logical value; should the resulting text start with *Lorem ipsum dolor sit amet*?

Details

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

Current implementation is very simple: words are selected randomly from a Zipf distribution (we base on a set of ca. 190 predefined Latin words). 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 `nparagraphs`.

See Also

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\n")
cat(stri_rand_lipsum(10), sep="\n\n")
```

---

stri_rand_shuffle *Randomly Shuffle Code Points in Each String*

Description

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

Usage

```
stri_rand_shuffle(str)
```

Arguments

`str`	character vector

Details

This operation may result in non-Unicode-normalized strings and may give strange output for bidirectional strings.

See also `stri_reverse` for a reverse permutation of code points.

Value

Returns a character vector.

See Also

Other random: stri_rand_lipsum, stri_rand_strings

Examples

```
stri_rand_shuffle(c("abcdefghi", "0123456789"))
# you can do better than this with stri_rand_strings:
stri_rand_shuffle(rep(stri_paste(letters, collapse=''), 10))
```

---

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

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 strange output 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.

See Also

Other random: stri_rand_lipsum, stri_rand_shuffle

Examples

```
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]')
))
```

---

stri_read_lines *[DRAFT API] Read Text Lines from a Text File*

Description

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

[THIS IS AN EXPERIMENTAL FUNCTION]

Usage

```
stri_read_lines(fname, encoding = "auto", locale = NA,
  fallback_encoding = stri_enc_get())
```

Arguments

`fname`	single string with file name
`encoding`	single string; input encoding, `"auto"` for automatic detection with `stri_enc_detect2`, and `NULL` or `""` for the current default encoding.
`locale`	single string passed to `stri_enc_detect2`; `NULL` or `""` for default locale, `NA` for checking just UTF-* family
`fallback_encoding`	single string; encoding to be used if encoding detection fails; defaults to the current default encoding, see `stri_enc_get`

Details

It is a substitute for the system's `readLines` function, with the ability to auto-detect input encodings (or to specify one manually), re-encode input without any strange function calls or sys options change, and split the text into lines with `stri_split_lines1` (which conforms with the Unicode guidelines for newline markers).

If `locale` is `NA` and auto-detection of UTF-32/16/8 fails, then `fallback_encoding` is used.

Value

Returns a character vector, with each line of text being a single string. The output is always in UTF-8.

See Also

Other files: stri_read_raw, stri_write_lines

---

stri_read_raw *[DRAFT API] Read Whole Text File as Raw*

Description

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

[THIS IS AN EXPERIMENTAL FUNCTION]

Usage

```
stri_read_raw(fname)
```

Arguments

fname	file name

Details

After reading a text file into memory (the vast majority of them will fit into RAM without any problems), you may perform encoding detection (cf. stri_enc_detect2), conversion (cf. stri_encode), and for example split it into text lines with stri_split_lines1.

Value

Returns a raw vector.

See Also

Other files: stri_read_lines, stri_write_lines

stri_remove_empty *Remove all empty strings from a character vector*

Description

This function removes all empty strings from a character vector.

Usage

```
stri_remove_empty(x, na_empty = FALSE)
```

Arguments

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

Value

Always returns a character vector.

See Also

Other utils: stri_list2matrix, stri_na2empty

Examples

```
stri_remove_empty(stri_na2empty(c("a", NA, "", "b")))
stri_remove_empty(c("a", NA, "", "b"))
stri_remove_empty(c("a", NA, "", "b"), TRUE)
```

stri_replace_all *Replace Occurrences of a Pattern*

Description

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

Usage

```
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, merge = FALSE,
  vectorize_all = TRUE)

stri_replace_first_charclass(str, pattern, replacement)

stri_replace_last_charclass(str, pattern, replacement)

stri_replace_all_coll(str, pattern, replacement, vectorize_all = TRUE, ...,
  opts_collator = NULL)

stri_replace_first_coll(str, pattern, replacement, ..., opts_collator = NULL)

stri_replace_last_coll(str, pattern, replacement, ..., opts_collator = NULL)

stri_replace_all_fixed(str, pattern, replacement, vectorize_all = TRUE, ...,
  opts_fixed = NULL)

stri_replace_first_fixed(str, pattern, replacement, ..., opts_fixed = NULL)

stri_replace_last_fixed(str, pattern, replacement, ..., opts_fixed = NULL)

stri_replace_all_regex(str, pattern, replacement, vectorize_all = TRUE, ...,
  opts_regex = NULL)

stri_replace_first_regex(str, pattern, replacement, ..., opts_regex = NULL)

stri_replace_last_regex(str, pattern, replacement, ..., opts_regex = NULL)
```

Arguments

`str`	character vector with strings to search in
`replacement`	character vector with replacements for matched patterns
`...`	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 defining 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 replaced by a corresponding replacement string?; stri_replace_all_* only

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

Details

By default, all the functions are vectorized over str, pattern, replacement. Then these functions scan the input string for matches of the pattern. 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, the 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 (their numbering starts from 1). In order to treat the $ character literally, escape it with a backslash. Moreover, ${name} are used for named capture groups.

stri_replace, stri_replace_all, stri_replace_first, and stri_replace_last are convenience functions; they just call stri_replace_*_* variants, depending on the arguments used. Using the underlying stri_replace functions will result in code running slightly faster.

If you would like to get rid of e.g. whitespaces from the start or end of a string, see stri_trim.

Value

All the functions return a character vector.

See Also

Other search_replace: stri_replace_na, stri_trim_both, stringi-search

Examples

```
stri_replace_all_charclass("aaaa", "[a]", "b", merge=c(TRUE, FALSE))

stri_replace_all_charclass("a\nb\tc   d", "\\p{WHITE_SPACE}", " ")
stri_replace_all_charclass("a\nb\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, " .*? ", "#")
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}).(\\p{N})', '$2-$1')
stri_replace_all_regex(c("stringi R", "REXAMINE", "123"), '( R|R.)', ' r ')

## Not run:
# named capture groups available since ICU 55
stri_replace_all_regex("words 123 and numbers 456",
   "(?<numbers>[0-9]+)", "!${numbers}!")

## End(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+%c("quick", "brown", "fox")%s+%"\\b", c("slow",  "black", "bear"), vectorize_all=FALSE)
```

`stri_replace_na` *Replace Missing Values in a Character Vector*

Description

This function offers a convenient way to replace each `NA` in a character vector 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(rep` `str2`. It may be used e.g. wherever "plain R" `NA` handling is desired, see Examples.

Value

Returns a character vector.

See Also

Other search_replace: stri_replace_all, stri_trim_both, stringi-search

Examples

```
x <- c('test', NA)
stri_paste(x, 1:2)                          # "test1" NA
paste(x, 1:2)                               # "test 1" "NA 2"
stri_paste(stri_replace_na(x), 1:2, sep=' ') # "test 1" "NA 2"
```

---

stri_reverse *Reverse Each String*

Description

Reverses 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 strange output for bidirectional strings.

See also stri_rand_shuffle for a random permutation of code points.

Value

Returns a character vector.

Examples

```
stri_reverse(c("123", "abc d e f"))
stri_reverse("ZXY (\u0105\u0104123$^).")
stri_reverse(stri_trans_nfd('\u0105')) == stri_trans_nfd('\u0105') # A, ogonek -> agonek, A
```

stri_split *Split a String By Pattern Matches*

Description

These functions split each element of `str` into substrings. `pattern` indicates delimiters that separate the input into tokens. The input data between the matches become the fields themselves.

Usage

```
stri_split(str, ..., regex, fixed, coll, charclass)

stri_split_fixed(str, pattern, n = -1L, omit_empty = FALSE,
  tokens_only = FALSE, simplify = FALSE, ..., opts_fixed = NULL)

stri_split_regex(str, pattern, n = -1L, omit_empty = FALSE,
  tokens_only = FALSE, simplify = FALSE, ..., opts_regex = NULL)

stri_split_coll(str, pattern, n = -1L, omit_empty = FALSE,
  tokens_only = FALSE, simplify = FALSE, ..., opts_collator = NULL)

stri_split_charclass(str, pattern, n = -1L, omit_empty = FALSE,
  tokens_only = FALSE, simplify = FALSE)
```

Arguments

`str`	character vector with 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 defining 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 `NA`s (`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 a search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for default settings;

Details

Vectorized over `str`, `pattern`, `n`, and `omit_empty`.

If `n` is negative, then all pieces are extracted. Otherwise, if `tokens_only` is `FALSE` (this is the default, for compatibility with the **stringr** package), then `n-1` tokes 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 tokes are substituted with missing strings.

Empty search patterns are not supported. If you would like 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. Relying on one of those underlying functions will make your code run slightly faster.

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.

See Also

Other search_split: `stri_split_boundaries`, `stri_split_lines`, `stringi-search`

Examples

```
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_fixed(c("ab,c", "d,ef,g", ",h", ""), ",", omit_empty=NA, simplify=NA)

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
```

`stri_split_boundaries` *Split a String at Specific Text Boundaries*

Description

This function locates specific text boundaries (like character, word, line, or sentence boundaries) and splits strings at the indicated positions.

Usage

```
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 as generated with `stri_opts_brkiter`; `NULL` for the default break iterator, i.e. `line_break`

Details

Vectorized over `str` and `n`.

If `n` is negative (default), then all pieces are extracted. Otherwise, if `tokens_only` is `FALSE` (this is the default, for compatibility with the **stringr** package), then `n-1` tokes 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 the 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.

See Also

Other search_split: stri_split_lines, stri_split, stringi-search

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

Other text_boundaries: stri_count_boundaries, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_brkiter, stri_split_lines, stri_trans_tolower, stri_wrap, stringi-search-boundaries, stringi-search

Examples

```
test <- "The\u00a0above-mentioned    features are very useful. " %s+%
   "Warm thanks to their developers. 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")

# 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 into text lines.

Usage

```
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 during splitting. If it is set to TRUE, then empty strings will never appear in the resulting vector.

Newlines are represented on different platforms e.g. by carriage return (CR, 0x0D), line feed (LF, 0x0A), CRLF, or next line (NEL, 0x85). Moreover, the Unicode Standard defines two unambiguous separator characters, Paragraph Separator (PS, 0x2029) and Line Separator (LS, 0x2028). Sometimes also vertical tab (VT, 0x0B) and form feed (FF, 0x0C) are used. These 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 is used to split a text line. For efficiency reasons, the search here is not performed by the regex engine, however.

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), thus it returns a character vector. Moreover, if the input string ends at a newline sequence, the last empty string is omitted from the result. Therefore, this function may be handy if you wish to split a loaded text file into text lines.

References

Unicode Newline Guidelines – Unicode Technical Report #13, http://www.unicode.org/standard/reports/tr13/tr13-5.html

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

See Also

Other search_split: stri_split_boundaries, stri_split, stringi-search

Other text_boundaries: stri_count_boundaries, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_brkiter, stri_split_boundaries, stri_trans_tolower, stri_wrap, stringi-search-boundaries, stringi-search

stri_startswith *Determine if the Start or End of a String Matches a Pattern*

Description

These functions check if a string starts or ends with a pattern occurrence.

Usage

```
stri_startswith(str, ..., fixed, coll, charclass)

stri_endswith(str, ..., fixed, coll, charclass)

stri_startswith_fixed(str, pattern, from = 1L, ..., opts_fixed = NULL)

stri_endswith_fixed(str, pattern, to = -1L, ..., opts_fixed = NULL)

stri_startswith_charclass(str, pattern, from = 1L)

stri_endswith_charclass(str, pattern, to = -1L)

stri_startswith_coll(str, pattern, from = 1L, ..., opts_collator = NULL)

stri_endswith_coll(str, pattern, to = -1L, ..., opts_collator = NULL)
```

Arguments

`str`	character vector
`...`	supplementary arguments passed to the underlying functions, including additional settings for `opts_collator`, `opts_fixed`, and so on.
`pattern, fixed, coll, charclass`	character vector defining search patterns; for more details refer to stringi-search
`from`	integer vector
`to`	integer vector
`opts_collator, opts_fixed`	a named list used to tune up a search engine's settings; see `stri_opts_collator` and `stri_opts_fixed`, respectively; `NULL` for default settings;

Details

Vectorized over `str`, `pattern`, and `from` or `to`.

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

Argument `start` controls the start position in `str` at which the `pattern` is being matched. On the other hand, `to` gives the end position.

Indices given by `from` or `to` are 1-based, i.e., an index equal to 1 denotes the first character in a string, which gives a typical R look-and-feel.

For negative indices 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 "$" metacharacters, see stringi-search-regex.

Value

Each function returns a logical vector.

See Also

Other search_detect: `stri_detect`, `stringi-search`

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")
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

Any of the strings must not 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**).

See Also

Other stats: stri_stats_latex

Examples

```
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)
```

---

stri_stats_latex *Statistics for a Character Vector Containing LaTeX Commands*

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

```
stri_stats_latex(str)
```

Arguments

str	character vector to be aggregated

Details

We use a slightly modified LaTeX Word Count algorithm taken from Kile 2.1.3, see http://kile.sourceforge.net/team.php for 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**).

See Also

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)
```

`stri_sub` *Extract a Substring From or Replace a Substring In a Character Vector*

Description

The first function extracts substrings under code point-based index ranges provided. The second one allows to substitute parts of a string with given strings.

Usage

```
stri_sub(str, from = 1L, to = -1L, length)

stri_sub(str, from = 1L, to = -1L, length, omit_na=FALSE) <- value
```

Arguments

str	character vector
from	integer vector or two-column matrix
to	integer vector; mutually exclusive with length and from being a matrix
length	integer vector; mutually exclusive with to and from being a matrix
omit_na	single logical value; if TRUE, missing values in any of the arguments provided will result in an unchanged input; replacement function only
value	character vector to be substituted with; replacement function only

Details

Vectorized over str, [value], from and (to or length). to and length are mutually exclusive.

to has priority over length. If from is a two-column matrix, then the first column is used as from and the second one as to. In such case arguments to and length are ignored.

Of course, the indices are code point-based, and not byte-based. Note that for some Unicode strings, the extracted substrings may not be well-formed, especially if the input is not NFC-normalized (see stri_trans_nfc), includes byte order marks, Bidirectional text marks, and so on. Handle with care.

Indices are 1-based, i.e., an index equal to 1 denotes the first character in a string, which gives a typical R look-and-feel. Argument to defines the last index of the substring, inclusive.

For negative indices 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.

In stri_sub, out-of-bound indices are silently corrected. If from > to, then an empty string is returned.

In stri_sub<-, some configurations of indices may work as string concatenation at the front, back, or middle.

Value

stri_sub returns a character vector. stri_sub<- changes the str object.

The extract function stri_sub returns the indicated substrings. The replacement function stri_sub<- is invoked for its side effect: after a call, str is modified.

See Also

Other indexing: stri_locate_all_boundaries, stri_locate_all

Examples

```
s <- "Lorem ipsum dolor sit amet, consectetur adipisicing elit."
stri_sub(s, from=1:3*6, to=21)
stri_sub(s, from=c(1,7,13), length=5)
stri_sub(s, from=1, length=1:3)
stri_sub(s, -17, -7)
stri_sub(s, -5, length=4)
(stri_sub(s, 1, 5) <- "stringi")
```

```
(stri_sub(s, -6, length=5) <- ".")
(stri_sub(s, 1, 1:3) <- 1:2)

x <- c("a;b", "c:d")
(stri_sub(x, stri_locate_first_fixed(x, ";"), omit_na=TRUE) <- "_")
```

---

stri_subset *Select Elements that Match a Given Pattern*

Description

These functions return or modify a subvector consisting of strings that match 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

```
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 with strings to search in
`...`	supplementary arguments passed to the underlying functions, including additional settings for `opts_collator`, `opts_regex`, `opts_fixed`, and so on
`value`	character vector to be substituted with; replacement function only

`pattern, regex, fixed, coll, charclass`
: character vector defining search patterns; for more details refer to stringi-search; the replacement functions accept only one pattern at a time

`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 a search engine's settings; see `stri_opts_collator`, `stri_opts_fixed`, and `stri_opts_regex`, respectively; `NULL` for default settings;

Details

Vectorized over `str`, and `pattern` or `value` (replacement version).

`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. Relying on these underlying functions will make your code run slightly faster.

Value

The `stri_subset` functions return a character vector. As usual, the output encoding is always UTF-8.

The `stri_subset<-` functions change the `str` object.

See Also

Other search_subset: `stringi-search`

Examples

```
stri_subset_regex(c("stringi R", "123", "ID456", ""), "^[0-9]+$")

x <- c("stringi R", "123", "ID456", "")
stri_subset_regex(x, "[^0-9]+|^$") <- NA
print(x)

x <- c("stringi R", "123", "ID456", "")
stri_subset_regex(x, "^[0-9]+$", negate=TRUE) <- NA
print(x)
```

`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. On the other hand, `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

```
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.

References

TimeZone class – ICU API Documentation, http://www.icu-project.org/apiref/icu4c/classicu_1_1TimeZone.html

See Also

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

```
## 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"

Details

With this function you may fetch some 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).

See Also

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

```
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 at which 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's in the northern hemisphere, and in the second half of the year it's in the southern hemisphere.
2. The precise date and time during the year when DST ends. In the first half of the year it's in the southern hemisphere, and in the second half of the year it's 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.

References

TimeZone class – ICU API Documentation, http://www.icu-project.org/apiref/icu4c/classicu_1_1TimeZone.html

ICU 4.8 Time Zone Names. http://site.icu-project.org/design/formatting/timezone/icu-4-8-time-zone-names

ICU TimeZone classes – ICU User Guide, http://userguide.icu-project.org/datetime/timezone

Date/Time Services – ICU User Guide, http://userguide.icu-project.org/datetime

See Also

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_info

Other timezone: stri_timezone_get, stri_timezone_info

Examples

```
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 info 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

```
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, last corresponding replacement code point is used.

Value

Returns a character vector.

See Also

Other transform: `stri_trans_general`, `stri_trans_list`, `stri_trans_nfc`, `stri_trans_tolower`

Examples

```
stri_trans_char("id.123", ".", "_")
stri_trans_char("babaab", "ab", "01")
```

---

stri_trans_general *General Text Transforms, Including Transliteration*

Description

ICU General transforms provide a general-purpose package for processing Unicode text. They are a powerful and flexible mechanism for handling a variety of different tasks, including:

- Upper Case, Lower Case, Title Case, Full/Halfwidth conversions,
- Normalization,
- Hex and Character Name conversions,
- Script to Script conversion/transliteration.

Usage

```
stri_trans_general(str, id)
```

Arguments

`str`	character vector
`id`	a single string with transform identifier, see `stri_trans_list`

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, the services performed here represent a much more general mechanism capable of handling a much broader range of tasks. In particular, the Transforms include pre-built 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;`

Value

Returns a character vector.

References

General Transforms – ICU User Guide, http://userguide.icu-project.org/transforms/general

See Also

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("tato nie wraca ranki wieczory", "pl-pl_FONIPA")
stri_trans_general("\u2620", "any-name") # character name
stri_trans_general("\\N{latin small letter a}", "name-any") # decode name
stri_trans_general("\u2620", "hex") # to hex
```

---

stri_trans_list *List Available Text Transforms and Transliterators*

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.

References

General Transforms – ICU User Guide, http://userguide.icu-project.org/transforms/general

See Also

Other transform: stri_trans_char, stri_trans_general, stri_trans_nfc, stri_trans_tolower

---

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 RFC\#5198.

As usual in **stringi**, if the input character vector is in the native encoding, it will be converted to UTF-8 automatically.

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).

On the other hand, `stri_trans_isnf*` return a logical vector.

References

Unicode Normalization Forms – Unicode Standard Annex #15, http://unicode.org/reports/tr15

Unicode Format for Network Interchange – RFC\#5198, http://tools.ietf.org/rfc/rfc5198.txt

Character Model for the World Wide Web 1.0: Normalization – W3C Working Draft, http://www.w3.org/TR/charmod-norm/

Normalization – ICU User Guide, http://userguide.icu-project.org/transforms/normalization (technical details)

Unicode Equivalence – Wikipedia, http://en.wikipedia.org/wiki/Unicode_equivalence

See Also

Other transform: `stri_trans_char`, `stri_trans_general`, `stri_trans_list`, `stri_trans_tolower`

Examples

```
stri_trans_nfd("\u0105") # Polish a with ogonek -> a, ogonek
stri_trans_nfkc("\ufdfa") # 1 codepoint -> 18 codepoints
```

`stri_trans_tolower` *Transform String with Case Mapping*

Description

These functions transform strings either to lower case, UPPER CASE, or to Title Case.

Usage

```
stri_trans_tolower(str, locale = NULL)

stri_trans_toupper(str, locale = NULL)

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 as generated with `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. In general,

- case mapping can change the number of code points and/or code units of a string,
- is language-sensitive (results may differ depending on 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 a 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.

For more general (but not locale dependent) text transforms refer to `stri_trans_general`.

Value

Each function returns a character vector.

References

Case Mappings – ICU User Guide, http://userguide.icu-project.org/transforms/casemappings

See Also

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

Other transform: stri_trans_char, stri_trans_general, stri_trans_list, stri_trans_nfc

Other text_boundaries: stri_count_boundaries, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_brkiter, stri_split_boundaries, stri_split_lines, stri_wrap, stringi-search-boundaries, stringi-search

Examples

```
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_totitle("GOOD-OLD cOOkiE mOnSTeR IS watCHinG You. Here HE comes!") # word boundary
stri_trans_totitle("GOOD-OLD cOOkiE mOnSTeR IS watCHinG You. Here HE comes!", 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 get rid of unnecessary whitespaces from strings. Trimming ends at the first or starts at the last `pattern` match.

Usage

```
stri_trim_both(str, pattern = "\\P{Wspace}")

stri_trim_left(str, pattern = "\\P{Wspace}")

stri_trim_right(str, pattern = "\\P{Wspace}")

stri_trim(str, side = c("both", "left", "right"), pattern = "\\P{Wspace}")
```

Arguments

str	a character vector of strings to be trimmed
pattern	a single pattern, specifying character classes that should be preserved (see stringi-search-charclass). Defaults to '\P{Wspace}.
side	character [stri_trim only]; defaults to "both"

Details

Vectorized over str and pattern.

stri_trim is a wrapper, which calls stri_trim_left or stri_trim_right as appropriate. It's slightly slower than trim_left or trim_right, and so shouldn't be used except for convenience.

Contrary to many other string processing libraries, our trimming functions are quite general. A character class, given by pattern, may be adjusted to suit your needs (most often you will use the default value). On the other hand, for replacing pattern matches with arbitrary replacement string, see stri_replace.

Interestingly, with these functions you may sometimes extract data, which in some cases require using regular expressions. E.g. you may get "23.5" out of "total of 23.5 bitcoins".

For trimming whitespaces, please note the difference between Unicode binary property '\p{Wspace}' (more general) and general character category '\p{Z}', see stringi-search-charclass.

Value

All these functions return a character vector.

See Also

Other search_replace: stri_replace_all, stri_replace_na, stringi-search

Other search_charclass: stringi-search-charclass, stringi-search

Examples

```
stri_trim_left("              aaa")
stri_trim_right("rexamine.com/", "\\p{P}")
stri_trim_both(" Total of 23.5 bitcoins. ", "\\p{N}")
stri_trim_both(" Total of 23.5 bitcoins. ", "\\p{L}")
```

---

stri_unescape_unicode *Unescape All Escape Sequences*

Description

Unescapes all known escape sequences

Usage

```
stri_unescape_unicode(str)
```

Arguments

str	character vector

Details

Uses **ICU** facilities to unescape Unicode character sequences.

The following ASCII standard escapes are recognized: \a, \b, \t, \n, \v, \?, \e, \f, \r, \", \', \\.

Moreover, the function understands the following ones: \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 UTF8 byte sequences.

Note that some versions of R on Windows cannot handle characters defined with \UXXXXXXXX. We are working on that.

Value

Returns a character vector. If an escape sequence is ill-formed, result will be NA and a warning will be given.

See Also

Other escape: stri_escape_unicode

Examples

```
stri_unescape_unicode("a\\u0105!\\u0032\\n")
```

---

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 as generated with 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.

References

Collation - ICU User Guide, http://userguide.icu-project.org/collation

See Also

Other locale_sensitive: %s<%, stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

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 should utilize to print a string with a monospaced 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 basing on http://www.cl.cam.ac.uk/~mgk25/ucs/wcwidth.c and the 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. 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.

References

East Asian Width – Unicode Standard Annex #11, http://www.unicode.org/reports/tr11/

See Also

Other length: stri_isempty, stri_length, stri_numbytes

Examples

```
stri_width(LETTERS[1:5])
nchar(stri_trans_nfkd("\u0105"), "width") # provides incorrect information
stri_width(stri_trans_nfkd("\u0105"))
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
```

---

stri_wrap *Word Wrap Text to Format Paragraphs*

Description

This function breaks text paragraphs into lines, of total width - if it is possible - of at most given width.

Usage

```
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 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 whitespaces? 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 at which a line break is possible. This is a locale-dependent operation. Otherwise, the breaks are only at whitespaces.

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

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 at error. You should split the input text into lines or e.g. substitute line breaks with spaces before applying this function.

On the other hand, 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 mode 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.

References

D.E. Knuth, M.F. Plass, Breaking paragraphs into lines, *Software: Practice and Experience* 11(11), 1981, pp. 1119–1184

See Also

Other locale_sensitive: `%s<%`, `stri_compare`, `stri_count_boundaries`, `stri_duplicated`, `stri_enc_detect2`, `stri_extract_all_boundaries`, `stri_locale_all_boundaries`, `stri_opts_collator`, `stri_order`, `stri_split_boundaries`, `stri_trans_tolower`, `stri_unique`, `stringi-locale`, `stringi-search-boundaries`, `stringi-search-coll`

Other text_boundaries: `stri_count_boundaries`, `stri_extract_all_boundaries`, `stri_locate_all_boundaries`, `stri_opts_brkiter`, `stri_split_boundaries`, `stri_split_lines`, `stri_trans_tolower`, `stringi-search-boundarie`
`stringi-search`

Examples

```
s <- stri_paste(
   "Lorem ipsum dolor sit amet, consectetur adipisicing 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 *[DRAFT API] Write Text Lines to a Text File*

Description

Writes a text file such that each element of a given character vector becomes a separate text line.

[THIS IS AN EXPERIMENTAL FUNCTION]

Usage

```
stri_write_lines(str, fname, encoding = "UTF-8",
  sep = ifelse(.Platform$OS.type == "windows", "\r\n", "\n"))
```

Arguments

str	character vector
fname	file name
encoding	output encoding, NULL or "" for the current default one
sep	newline separator

Details

It is a substitute for the R writeLines function, with the ability to re-encode output without any strange function calls.

Note that we suggest using the UTF-8 encoding for all text files: thus, it is the default one for the output.

Value

This function does not return anything interesting

See Also

Other files: stri_read_lines, stri_read_raw

`%s<%` *Compare Strings with or without Collation*

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 %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 default collator options. Thus, they are vectorized over e1 and e2.

%stri==% tests for canonical equivalence of strings (see stri_cmp_equiv) and is a locale-dependent operation. On the other hand, %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.

See Also

Other locale_sensitive: stri_compare, stri_count_boundaries, stri_duplicated, stri_enc_detect2, stri_extract_all_boundaries, stri_locate_all_boundaries, stri_opts_collator, stri_order, stri_split_boundaries, stri_trans_tolower, stri_unique, stri_wrap, stringi-locale, stringi-search-boundaries, stringi-search-coll

Examples

```
"a" %stri<% "b"
c("a", "b", "c") %stri>=% "b"
```

%s+% *Concatenate Two Character Vectors*

Description

Binary operators for joining (concatenating) two character vectors, with a typical R look-and-feel.

Usage

```
e1 %s+% e2

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 %+% e2 %+% 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.

Examples

```
c('abc', '123', '\u0105\u0104') %stri+% letters[1:6]
'ID_' %stri+% 1:5
```

Index