Package ‘tidycjk’

October 12, 2026

Type Package

Title Tidy Tools for Chinese, Japanese and Korean Text

Version 0.2.0

Description A tidy toolkit for text that is written in Chinese, Japanese or Korean. Most text tooling in R assumes that words are separated by whitespace, which CJK writing does not use, so ordinary summaries of a text column either treat a sentence as one undifferentiated blob or split it into isolated characters. Word segmentation is a pluggable engine, with dictionary-based segmentation for Chinese and Japanese supplied through 'ICU' and further engines registrable by the caller, because where a word ends is a fact about a language and not about Unicode. 'tidycjk' classifies characters by Unicode block, reports which script and which language a text is written in, measures how much of a text is CJK, and turns those measurements into tibbles that slot straight into a 'tidyverse' workflow. It also measures display width in terminal columns, pads, truncates and wraps to a width rather than to a character count, and normalises fullwidth and halfwidth forms surgically -- including composing halfwidth katakana voiced marks into single code points -- while offering the Unicode normalisation forms separately for when a full 'NFKC' fold is what is wanted. Romanisation, conversion between simplified and traditional Han, conversion between the two kana syllabaries, and decomposition of Hangul syllables into jamo are provided on the same vectorised contract. Sentence segmentation, character n-grams, punctuation removal by Unicode category and locale-aware ordering cover the preprocessing that generic tooling gets wrong on CJK, where the sentence terminators are ideographic, a code point sort is not an alphabetical one, and the POSIX punctuation class is resolved by the C library rather than by Unicode. Language detection deliberately returns NA rather than guessing when a text is written in Han characters only, because Japanese written without kana cannot be distinguished from Chinese by script alone. Everything is derived from the Unicode specification; the package makes no network requests and needs no compiled code of its own.

License GPL (>= 3)

URL https://pursuitofdatascience.github.io/tidycjk/, https://github.com/PursuitOfDataScience/tidycjk

BugReports https://github.com/PursuitOfDataScience/tidycjk/issues

Encoding UTF-8

Language en-GB

Depends R (>= 3.5.0)

Imports dplyr (>= 1.1.0), stringi, tibble, utils

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

RoxygenNote 7.3.2

NeedsCompilation no

Author Youzhi Yu [aut, cre]

Maintainer Youzhi Yu <yuyouzhi666@icloud.com>

Repository CRAN

Date/Publication 2026-10-11 22:30:02 UTC

Contents

cjk_blocks . . . 3
cjk_char_counts . . . 4
cjk_detect_language . . . 5
cjk_jamo . . . 7
cjk_ngrams . . . 9
cjk_normalize . . . 10
cjk_pad . . . 12
cjk_ratio . . . 14
cjk_romanize . . . 15
cjk_script . . . 17
cjk_segment . . . 18
cjk_segmenters . . . 19
cjk_sentences . . . 22
cjk_simplify . . . 23
cjk_sort . . . 25
cjk_strip_punct . . . 27
cjk_summary . . . 28
cjk_tokens . . . 29
cjk_truncate . . . 31
cjk_width . . . 32
cjk_wrap . . . 33
has_cjk . . . 36
to_halfwidth . . . 37
to_hiragana . . . 39

Index **41**

`cjk_blocks` *The Unicode blocks 'tidycjk' recognises*

Description

`cjk_blocks()` returns the block table that every other function in the package consults. It is the package's definition of "CJK", written down and exported so that it can be inspected rather than guessed at.

Usage

```
cjk_blocks()
```

Details

The table is sorted by `start` and contains no overlapping ranges, which is what allows a whole corpus of code points to be classified with a single `findInterval()` call rather than a per-character regular expression.

A block appears as more than one row where a single script label would be wrong for part of it. Halfwidth Katakana (U+FF65-U+FF9F) is a sub-range of the Halfwidth and Fullwidth Forms block; because the katakana label is the more useful one, the enclosing block appears as two rows either side of it. Kana Supplement, Kana Extended-A and Small Kana Extension each hold both hiragana and katakana, so each is split where the script changes. Otherwise the labels follow block boundaries rather than the Unicode Script property: the halfwidth Hangul jamo at U+FFA0-U+FFDC are reported as `"fullwidth"` rather than `"hangul"`, and the one hiragana digraph among the katakana of Kana Extended-A (U+1B123) as `"katakana"`. Use `cjk_char_counts()` if you need to see exactly which code points a text contains.

All eleven blocks of unified ideographs are covered: the base block and Extensions A through J. They are not in alphabetical order, because Unicode allocated Extension I (U+2EBF0) below Extension G (U+30000) rather than after Extension H, and this table is in code point order.

The table is current to Unicode 18.0. A block Unicode adds after that is not here until the table is updated, and `has_cjk()` answers `FALSE` for its characters in the meantime; that is how Extension J, added in Unicode 17.0, came to be missing from 0.1.0.

Ranges are Unicode block bounds, with one exception. Hangul Syllables stops at U+D7A3, the last assigned syllable, rather than at U+D7AF where the block ends; the twelve code points in between are unassigned, and calling them hangul would be reporting text that cannot exist. Elsewhere the block bound is used as-is, so a handful of unassigned code points inside a covered block (U+3100 to U+3104 at the head of Bopomofo, for instance) do count.

Each phonetic script is covered in full, extension blocks included: Bopomofo Extended (U+31A0-U+31BF) alongside Bopomofo, and the hentaigana, archaic and small kana and the Minnan tone letters of Kana Supplement, Kana Extended-A, Kana Extended-B and Small Kana Extension alongside Hiragana and Katakana. What is deliberately absent is everything that is neither a letter, an ideograph, CJK punctuation nor a width variant: the radical blocks (U+2E80-U+2EFF and the Kangxi radicals at U+2F00-U+2FDF), CJK Strokes (U+31C0-U+31EF) and the Ideographic Description Characters (U+2FF0-U+2FFF); the vertical presentation forms in Vertical Forms (U+FE10-U+FE1F)

and CJK Compatibility Forms (U+FE30-U+FE4F); and the parenthesised, circled and squared compatibility symbols in Enclosed CJK Letters and Months, CJK Compatibility and Enclosed Ideographic Supplement. Those are typographic presentation forms rather than text, and counting them as CJK would inflate `cjk_ratio()` on a column that contains no CJK writing at all. Also outside are the specialist marks of Ideographic Symbols and Punctuation (U+16FE0-U+16FFF), and the two bopomofo tone marks Unicode placed in Spacing Modifier Letters, U+02EA and U+02EB, far from any CJK block.

The `script` column takes one of eight values. Six of them name a writing system – `"han"`, `"hiragana"`, `"katakana"`, `"hangul"`, `"bopomofo"`, `"kanbun"` – and two are ancillary: `"punctuation"` for the CJK Symbols and Punctuation block, and `"fullwidth"` for the width-variant forms. Only the six writing systems are consulted by `cjk_detect_language()`.

Value

A tibble with one row per block and columns `block` (the Unicode block name), `script` (tidycjk's script label), `start` and `end` (the inclusive code point bounds, as integers) and `n_codepoints`.

See Also

`cjk_script()` for the dominant script of a string, `cjk_char_counts()` for a per-character breakdown.

Examples

```
cjk_blocks()

# the blocks that make up "han"
subset(cjk_blocks(), script == "han")
```

`cjk_char_counts` *Count the CJK characters in a text column*

Description

`cjk_char_counts()` returns one row per distinct CJK character in a text column, with the script and Unicode block it belongs to and how often it occurred. It is the frequency table you would otherwise write by hand before every CJK analysis.

Usage

```
cjk_char_counts(data, col)
```

Arguments

`data`	A data frame or tibble containing a text column.
`col`	The text column to scan, supplied unquoted. A non-character column is coerced with `as.character()`; see `has_cjk()` for why that makes a numeric column a poor thing to measure. A list-column is an error rather than a coercion, for the reason given there.

Details

Only CJK characters appear; Latin letters, digits and whitespace are dropped. "CJK" means any block listed by `cjk_blocks()`, so ideographic punctuation and fullwidth forms are included and are labelled as such in `script` – which makes this the quickest way to find out that a column you thought was clean is full of fullwidth spaces.

Rows are ordered by descending count, and ties are broken by first appearance in the column rather than by the session's collation, so the output does not change with the locale.

Grouping is ignored: the counts are always for the whole column. Use `dplyr::group_modify()` if you need them per group.

Value

A tibble with one row per distinct character and columns `char`, `codepoint` (integer), `script`, `block` and `n`. A column with no CJK in it gives a zero-row tibble with those columns.

See Also

`cjk_summary()` for the column-level figures, `cjk_blocks()` for the block table these labels come from.

Examples

```
# "Chinese writing" twice over, then "the Japanese language" in kanji and
# kana, then a row with no CJK at all
df <- data.frame(
  text = c("\u4e2d\u6587\u4e2d\u6587",
           "\u65e5\u672c\u306e\u3053\u3068\u3070",
           "ascii only")
)
cjk_char_counts(df, text)
```

`cjk_detect_language` *Which language is the text written in?*

Description

`cjk_detect_language()` infers the language of each string from the scripts it contains, and returns `NA` when the scripts do not settle the question.

Usage

```
cjk_detect_language(x, han_only = NA_character_)
```

Arguments

x	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
han_only	What to return for a string written in Han characters only. Defaults to `NA_character_`, meaning "undecidable". Set it to `"chinese"` to opt into the guess.

Details

The rules are applied in order:

1. Hiragana or katakana present: `"japanese"`. Only Japanese uses kana. The katakana middle dot U+30FB, and its halfwidth form U+FF65, do not count: they are punctuation that Chinese and Korean text use too, to separate the parts of a transliterated name.
2. Kanbun annotation marks present: `"japanese"`. They exist to make Classical Chinese readable as Japanese.
3. Hangul present: `"korean"`.
4. Bopomofo present: `"chinese"`. Bopomofo annotates Mandarin.
5. Han characters and nothing else from the list above: `NA` by default.
6. No CJK writing system at all: `NA`.

Rule 5 is the point of the function. Japanese written without kana – a headline, a shop sign, a personal name, a compound such as U+6771 U+4EAC U+90FD (Tokyo Metropolis) – is not distinguishable from Chinese by script alone, because both are writing the same Han characters. Any package that answers `"chinese"` there is guessing, and it will be wrong on Japanese input in a way the caller cannot detect. `tidycjk` returns `NA` instead.

If your corpus is known to be Chinese and you want the guess anyway, ask for it explicitly with `han_only = "chinese"`. Making it an argument keeps the assumption in the script, where a reader of the analysis can see it.

CJK punctuation and fullwidth forms are ignored here, even though `has_cjk()` counts them, because all three languages share them.

Value

A character vector the same length as `x`, holding `"japanese"`, `"korean"`, `"chinese"`, the value of `han_only`, or `NA`.

See Also

`cjk_script()`, which reports what is actually there rather than inferring from it.

Examples

```
# kana settles it; hangul settles it
cjk_detect_language(c("\u3053\u3093\u306b\u3061\u306f", "\uc548\ub155"))

# U+6771 U+4EAC U+90FD is Tokyo Metropolis: Japanese, written without kana,
# and therefore indistinguishable from Chinese. The honest answer is NA.
cjk_detect_language("\u6771\u4eac\u90fd")

# opt in to the guess when you know the corpus is Chinese
cjk_detect_language("\u6771\u4eac\u90fd", han_only = "chinese")
```

`cjk_jamo` *Decompose and recompose Hangul syllables*

Description

`cjk_jamo()` splits each Hangul syllable into the jamo it is built from; `cjk_compose_jamo()` puts them back together.

Usage

```
cjk_jamo(x)

cjk_compose_jamo(x)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

A modern Hangul syllable is a composite. Unicode encodes 11,172 of them precomposed in the Hangul Syllables block, each one algorithmically derived from a leading consonant, a vowel and an optional trailing consonant: `SIndex = (LIndex * 21 + VIndex) * 28 + TIndex`. Both verbs apply that arithmetic directly, so the decomposition is exact: no table can be out of date and no syllable is missed.

Value

`cjk_jamo()` returns a list the same length as `x`, each element a character vector of jamo and of the other characters, one per code point; `NA` gives `NA_character_` and the empty string gives `character(0)`. `cjk_compose_jamo()` takes a character vector and returns one.

Only Hangul is touched

A character that is not a Hangul syllable passes through `cjk_jamo()` as itself, and `cjk_compose_jamo()` joins conjoining jamo into syllables and changes nothing else, so both are safe to run over a mixed column. That is deliberately narrower than Unicode normalisation. NFD would also split an accented Latin letter, and NFD and NFC alike replace a CJK compatibility ideograph with its unified form, which in a Korean column means silently rewriting Hanja: the compatibility ideographs exist so that the Korean legacy encodings round-trip. Use `cjk_normalize()` when normalisation is what you want.

The round trip

`cjk_jamo()` returns a **list**, one character vector of jamo per element of `x`, and `cjk_compose_jamo()` takes a character vector, so the two do not compose directly: join each element first, as the `rt()` helper in the examples does. Writing `cjk_compose_jamo(cjk_jamo(x))` is an error rather than a silent wrong answer, because `as.character()` would deparse the list and hand back the literal string `c("\u1112", "\u1161", "\u11ab")`.

Joined and composed, the jamo give back `x` exactly, with one exception that is the point of `cjk_compose_jamo()`: conjoining jamo that `x` already held as separate code points come back composed into their syllable.

That makes jamo the right unit for questions the syllable hides: which initial consonants a corpus favours, whether two spellings differ only in a final consonant, or how to sort by consonant. U+D55C counts three ways, each right for a different question: one character to `nchar()`, two terminal columns to `cjk_width()` (a Hangul syllable is East Asian Wide), and three jamo here.

See Also

`cjk_script()` to detect Hangul, `cjk_blocks()` for the blocks involved, `cjk_normalize()` for the Unicode normalisation forms.

Examples

```
# U+D55C U+AE00, "Hangul"
cjk_jamo("\ud55c\uae00")

# join each element and compose it: the round trip is exact
rt <- function(x) {
  cjk_compose_jamo(vapply(cjk_jamo(x), paste, character(1), collapse = ""))
}
rt("\ud55c\uae00")

# text that is not Hangul is left alone: the accent stays where it was,
# and the compatibility ideograph U+F900 is not swapped for U+8C48
cjk_jamo("e\u0301")
rt("\uf900") == "\uf900"
```

`cjk_ngrams` *Character n-grams*

Description

Returns every run of `n` consecutive characters in each string. Character n-grams are the standard dictionary-free baseline for Chinese information retrieval and text classification, and for many tasks they are competitive with word segmentation while needing no model at all.

Usage

```
cjk_ngrams(x, n = 2L)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`n`	Size of each gram, a single positive whole number. Defaults to `2`, the usual choice for Chinese.

Details

For a language that writes word boundaries, n-grams are a crude feature. For Chinese they are not: most words are one or two characters, so character bigrams capture the majority of them without a dictionary, and they degrade gracefully on the out-of-vocabulary terms – names, slang, new coinages – where a segmenter is least reliable. Compare against `cjk_segment(engine = "icu")` on your own corpus rather than assuming either wins.

Characters are counted as code points, the same unit `cjk_ratio()` uses, so a supplementary-plane ideograph counts once. A combining mark or a variation selector is a code point too, and therefore its own position in the window: a base character followed by a selector contributes two grams rather than one. That is consistent with the rest of the package, and it is worth knowing if your corpus carries ideographic variation sequences.

Value

A list the same length as `x`, each element a character vector of n-grams in order of occurrence, with repeats kept. `NA` gives `NA_character_`; the empty string gives `character(0)`.

Whitespace ends a window

An n-gram is never formed across a space, so no gram spans two words of a Latin run or two sides of an ideographic space U+3000. Punctuation is kept, because in CJK it is often informative and it is cheap to filter afterwards; whitespace is not, because a gram containing one is an artefact of layout.

A string shorter than `n` characters yields `character(0)` rather than a padded gram.

See Also

`cjk_segment()` for word tokens, `cjk_sentences()` for sentences.

Examples

```
# U+4E2D U+6587 U+5F88 U+597D, "Chinese is good"
cjk_ngrams("\u4e2d\u6587\u5f88\u597d")

# trigrams, and the short-string case
cjk_ngrams("\u4e2d\u6587\u5f88\u597d", n = 3)
cjk_ngrams("\u4e2d")

# no gram is formed across the space
cjk_ngrams("\u4e2d\u6587 \u4f60\u597d")
```

`cjk_normalize` *Apply Unicode normalisation*

Description

`cjk_normalize()` puts text into one of the Unicode normalisation forms, so that strings which look the same compare the same. It optionally removes variation selectors first, which defeat an exact match on their own and which only one of the forms takes out.

Usage

```
cjk_normalize(
  x,
  form = c("nfc", "nfd", "nfkc", "nfkd", "nfkc_casefold"),
  drop_variation_selectors = FALSE
)
```

Arguments

`x` A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that

builds the list – so a list, a data frame or a function is an error naming what to do instead.

`form` One of `"nfc"`, `"nfd"`, `"nfkc"`, `"nfkd"` or `"nfkc_casefold"`. Defaults to `"nfc"`.

`drop_variation_selectors` Remove variation selectors before normalising. Defaults to `FALSE`.

Details

Two strings can render identically and still differ as data. Japanese U+304C is one code point or two depending on where the text came from; a compatibility ideograph and its unified equivalent are distinct code points for the same character; an ideographic variation sequence adds an invisible selector that only a font reacts to. Any of these will split one token into two in a frequency count, a join or a `group_by()`.

Value

A character vector the same length as `x`. `NA` input gives `NA`; the empty string gives the empty string.

Which form

`"nfc"` and `"nfd"` are canonical: they preserve the identity of every character and differ only in whether marks are composed. `"nfc"` is the form almost all text is already in and the right default.

`"nfkc"` and `"nfkd"` add compatibility folding, which is lossy by design. On CJK text it narrows fullwidth ASCII, widens halfwidth katakana, turns the ideographic space U+3000 into a plain space, and unpacks circled numbers and squared kana – U+3314 becomes the two characters U+30AD U+30ED. That is the right trade for search and token matching and the wrong one for anything that has to be shown back to a reader. `"nfkc_casefold"` is `"nfkc"` plus case folding, for case-insensitive matching of the Latin mixed into CJK text.

`"nfkc_casefold"` also deletes every code point Unicode marks `Default_Ignorable`, which the other four forms all keep. That is the variation selectors below, and with them the zero-width joiner and non-joiner, the zero-width space, the soft hyphen, the tag characters and U+FEFF – so a byte-order mark at the front of a string survives the other forms and does not survive this one. For matching that is the point. If you need the folding without the deletions, use `"nfkc"` and lowercase separately.

`to_halfwidth()` and `to_fullwidth()` are the narrow alternative: they move text along the width axis and touch nothing else.

NFC is not a no-op on CJK

Canonical normalisation is usually described as safe, and on Latin text it is. On Han it is not. Compatibility ideographs have *singleton* canonical mappings, so `"nfc"` **rewrites them just as** `"nfkc"` **does**: U+FA0C becomes U+5140 and U+F900 becomes U+8C48. Of the 472 characters in the CJK Compatibility Ideographs block, 460 are folded away by plain NFC; the 12 that survive – U+FA0E, U+FA0F, U+FA11, U+FA13, U+FA14, U+FA1F, U+FA21, U+FA23, U+FA24, U+FA27, U+FA28, U+FA29 – are unified ideographs that were encoded in that block by accident and have no mapping to apply. All 542 characters of the Supplement block are folded. So the block is not preserved and not uniformly folded either: if the distinction between a compatibility ideograph and its unified form carries meaning in your data, as it can in Korean and Japanese name records, normalising at all will destroy most of it. Keep the original column.

Variation selectors

`drop_variation_selectors = TRUE` removes them before the form is applied. They are removed first rather than afterwards because a selector blocks canonical composition across itself, so stripping one after normalising can leave text that is no longer in the form just requested.

The set removed is Unicode's `Variation_Selector` property, not a list written out here: the sixteen at U+FE00..U+FE0F, the 240 ideographic ones at U+E0100..U+E01EF, and the Mongolian free variation selectors – of which there are four on a current ICU and three on one built against Unicode 10.0, because U+180F arrived in Unicode 14.0. Taking the set from the property rather than from a literal means the verb follows whichever ICU is installed. Dropping them merges the glyph variants of a character into one token, which is what a frequency count wants and what a typesetter does not.

`"nfc"`, `"nfd"`, `"nfkc"` and `"nfkd"` all leave every one of them in place, so this argument is the only way to be rid of them under those forms. `"nfkc_casefold"` removes them regardless, as part of the wider deletion described above, and the argument is then redundant rather than wrong.

See Also

`to_halfwidth()` and `to_fullwidth()` for the width axis alone, `cjk_sort()` for comparison that ignores these differences without rewriting the text.

Examples

```
# the same syllable composed and decomposed, which nchar() can tell apart
both <- c("\u304c", "\u304b\u3099")
nchar(both)
nchar(cjk_normalize(both))

# NFKC folds the width axis and more besides; NFC leaves both alone
cjk_normalize("\uff21\uff22\u3000\u2460", form = "nfkc")
cjk_normalize("\uff21\uff22\u3000\u2460", form = "nfc")

# a compatibility ideograph is rewritten by plain NFC
sprintf("%X", utf8ToInt(cjk_normalize("\ufa0c")))

# an ideographic variation sequence survives every form until dropped
ivs <- "\u8fbb\U000E0100"
nchar(cjk_normalize(ivs))
nchar(cjk_normalize(ivs, drop_variation_selectors = TRUE))
```

---

cjk_pad *Pad text to a display width*

Description

`cjk_pad()` pads each string with a fill character until it occupies at least `width` terminal columns. Unlike a pad that counts characters, it produces columns that actually line up when the text is CJK.

Usage

```
cjk_pad(x, width, side = "right", pad = " ")
```

Arguments

x	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
width	Target display width in columns. Recycled against `x`; a pair of lengths that does not recycle cleanly is an error rather than a warning and a short result.
side	Which side to add padding to: `"right"` (the default, which left-aligns the text), `"left"` or `"both"`.
pad	A single character to pad with. Must be one column wide.

Value

A character vector the same length as the recycled inputs. Strings already at least `width` columns wide are returned unchanged – `cjk_pad()` never truncates. `NA` input, and an `NA` width, give `NA`.

See Also

`cjk_truncate()` for the other direction; `stringi::stri_pad()`, which this wraps.

Examples

```
# both strings end up six columns wide
cjk_pad(c("\u4e2d\u6587", "abcd"), 6)

# right-align instead
cjk_pad(c("\u4e2d\u6587", "abcd"), 6, side = "left")

# a pad that counts characters rather than columns: nchar() calls the two
# strings 2 and 4 long, so the CJK cell is handed four spaces and comes out
# eight columns wide. (formatC() and format() are column-aware and get this
# right; sprintf("%-6s") counts bytes and under-fills instead.)
pad_by_char <- function(x, n) paste0(x, strrep(" ", pmax(n - nchar(x), 0)))
cat(paste0("|", pad_by_char(c("\u4e2d\u6587", "abcd"), 6), "|"), sep = "\n")
cat(paste0("|", cjk_pad(c("\u4e2d\u6587", "abcd"), 6), "|"), sep = "\n")
```

`cjk_ratio` *What share of the text is CJK?*

Description

`cjk_ratio()` reports the proportion of each string's characters that fall in a CJK Unicode block, from 0 to 1. It is the natural way to find the rows of a mixed corpus that are actually CJK, as opposed to the ones carrying a single stray ideograph.

Usage

```
cjk_ratio(x)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

Characters are counted as Unicode code points, so an ideograph from a supplementary plane counts once, not twice. The denominator is every character in the string, including spaces and Latin punctuation.

Value

A numeric vector the same length as `x`, between 0 and 1. `NA` input gives `NA`. The empty string gives `NA` rather than 0, because the ratio is 0/0 and undefined.

See Also

`has_cjk()` for the yes/no version, `cjk_summary()` for the column-level summary.

Examples

```
# all CJK; two CJK characters of seven, because the space and the Latin
# letters count too; none; the empty string; NA
cjk_ratio(c("\u4e2d\u6587", "half \u4e2d\u6587", "none", "", NA))
```

`cjk_romanize` *Romanise CJK text*

Description

Transliterates Han, kana and Hangul into the Latin alphabet: pinyin for Chinese, romaji for Japanese kana, Revised-Romanisation-style output for Korean.

Usage

```
cjk_romanize(x, ascii = FALSE)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`ascii`	If `TRUE`, strip diacritics so the result is plain ASCII: pinyin tone marks are removed, so that `wo` with a caron becomes plain `wo`. Defaults to `FALSE`, which keeps them.

Details

The mapping is ICU's `Any-Latin`, which handles all three scripts in one pass, so mixed text does not need splitting first. Text that is already Latin is left alone.

Value

A character vector the same length as `x`. `NA` gives `NA`.

What this is not

Romanisation is not a function of the code point alone, and ICU treats it as though it were. Four consequences worth knowing before you trust the output:

- **Han is always read as Chinese.** ICU routes every Han character through pinyin, whatever language surrounds it, so Japanese kanji come back with Chinese readings: U+65E5 U+672C U+8A9E ("Japanese language") romanises to `ri ben yu`, not `nihongo`. This is not a near miss to be cleaned up afterwards – it is the wrong language. Do not use `cjk_romanize()` on Japanese kanji.

- **A character with two readings gets one.** ICU reads each Han character on its own, so a polyphonic character gets its most common reading whatever word it is in: U+94F6 U+884C ("bank") romanises to `yin xing` rather than `yin hang`, and U+97F3 U+4E50 ("music") to `yin le` rather than `yin yue` (tone marks omitted here). Treat the output as a sort or search key, not as pinyin to show a reader.
- **No word spacing.** ICU inserts a space between Han syllables but not between scripts, so Han followed immediately by kana romanises to a run with no boundary where the script changes. Segment first with `cjk_segment()` if you need words.
- **Kana particles are spelled, not pronounced.** The Japanese topic particle U+306F romanises to `ha`, which is how it is written and not how it is said.

It is slow, by a wide margin

ICU's transliterator is the most expensive thing this package calls. On a column of 5,000 twenty-character Chinese documents, romanising ran at a median of about 45,000 characters a second (95% CI 43,500 to 45,600) and took about 27 times as long as `cjk_segment(engine = "icu")` on the same column (95% CI 24 to 29; the closest of the pairs was 17 times). The cost is ICU's: a bare `stringi::stri_trans_general(x, "Any-Latin")` takes the same time (ratio 1.01, 95% CI 0.99 to 1.02), and there is no faster route to the same answer.

It also gets slower per character as one string gets longer, so the same 100,000 characters took 1.4 times as long pasted into a single string as kept as a column (95% CI 1.35 to 1.43). Keep a corpus as one row per document, and romanise once into a column you keep rather than inside a loop.

Those figures are medians of 30 paired, interleaved runs on one machine (an AMD EPYC 7702, ICU 74.1) and will move with the CPU, the load and the ICU build; the ratios are the durable part. Nothing in the test suite asserts them, because a timing assertion on a CRAN build machine fails for reasons that have nothing to do with this package.

For Japanese specifically, a morphological analyser that knows the reading is the right tool: gibasa, which binds MeCab. This function is for Chinese, for kana, and for getting a sortable ASCII key out of a CJK column.

See Also

`cjk_simplify()` for Han script conversion, `cjk_segment()` for word boundaries.

Examples

```
# U+4E2D U+6587, "Chinese writing"
cjk_romanize("\u4e2d\u6587")            # pinyin, with tone marks
cjk_romanize("\u4e2d\u6587", ascii = TRUE)

# Han is read as Chinese even in Japanese text -- see Details
cjk_romanize("\u65e5\u672c\u8a9e")

# kana and Hangul go through the same call
cjk_romanize(c("\u3053\u3093\u306b\u3061\u306f", "\uc548\ub155"))
```

cjk_script — *Which CJK script dominates the text?*

Description

`cjk_script()` returns the script that accounts for the most CJK characters in each string: one of `"han"`, `"hiragana"`, `"katakana"`, `"hangul"`, `"bopomofo"`, `"kanbun"`, `"punctuation"` or `"fullwidth"`.

Usage

```
cjk_script(x)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

Only CJK characters vote. Latin letters, digits and whitespace are ignored entirely, so a string of English with two ideographs in it is `"han"` rather than something averaged over the whole string.

Ties are broken by first appearance in the string – not alphabetically and not by the session's collation – so the result never depends on the locale.

Value

A character vector the same length as `x`. Strings with no CJK characters – and `NA` strings, and `""` – give `NA`.

See Also

`cjk_blocks()` for the script labels, `cjk_char_counts()` for the full per-character breakdown rather than just the winner.

Examples

```
# Chinese, Japanese, Korean, then a string with no CJK at all
cjk_script(c("\u4e2d\u6587", "\u3053\u3093\u306b\u3061\u306f",
             "\uc548\ub155", "ascii"))

# mixed Han and kana: four kana outvote two ideographs
cjk_script("\u65e5\u672c\u306e\u3053\u3068\u3070")
```

`cjk_segment` *Split CJK text into words*

Description

`cjk_segment()` splits each string into tokens. Chinese and Japanese do not put spaces between words, so splitting on whitespace returns the whole sentence as one token; this dispatches to a segmentation engine instead.

Usage

```
cjk_segment(x, engine, ...)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`engine`	Name of a segmentation engine, or a function implementing one. Required; see `cjk_segmenters()`.
`...`	Passed to the engine. Name these so they are not a prefix of `engine` (or of `data`/`col` in `cjk_tokens()`); see "Passing arguments to an engine" in `cjk_segmenters()`.

Details

Two engines ship with the package. `"icu"` is a real word segmenter, using the dictionary ICU carries inside **stringi**, so it costs no dependency you have not already installed. `"character"` tokenises by character rather than by word.

`engine` is required and has no default, because `"character"` answers a different question from the one a caller asking for words is asking, and quietly returning it would be the mistake this package exists to avoid. The two also differ on punctuation: `"icu"` drops it, `"character"` keeps CJK punctuation as tokens. `cjk_segmenters()` lists what is available and shows how to register another.

Value

A list the same length as `x`, each element a character vector of tokens. `NA` input gives `NA_character_`; the empty string gives `character(0)`.

See Also

`cjk_tokens()` for the tidy version, `cjk_segmenters()` for the engines and for registering one.

Examples

```
# a real word segmenter: four words, not six characters
cjk_segment("\u6211\u4eca\u5929\u5f88\u958b\u5fc3", engine = "icu")

# the dictionary-free baseline, one token per CJK character
cjk_segment("\u6211\u4eca\u5929\u5f88\u958b\u5fc3", engine = "character")

# non-CJK runs stay whole and are split on whitespace
cjk_segment("hello \u4e2d\u6587 world", engine = "character")
```

`cjk_segmenters` *Segmentation engines*

Description

`cjk_segmenters()` lists the engines `cjk_segment()` can dispatch to, and `register_cjk_segmenter()` adds one.

Usage

```
cjk_segmenters()

register_cjk_segmenter(name, fn)
```

Arguments

`name`	Name of the engine, a single string.
`fn`	A function of `(x, ...)` returning a list of character vectors.

Details

Where a word begins and ends in CJK text is a fact about a language, not about Unicode, so it cannot be derived the way everything else in this package is. It needs a dictionary and a statistical model, and which one is right depends on the language and the corpus. **tidycjk** therefore dispatches on a name rather than committing to one.

Two engines ship with the package.

`"icu"` is a real word segmenter. ICU carries dictionary-based break iterators for Chinese and Japanese, and stringi carries ICU, so this costs no dependency you have not already installed: a six-character Chinese sentence comes back as its four words rather than as six characters.

It returns surface forms only, with no part of speech, lemma or user dictionary, and its models are lighter than MeCab's or jieba's tuned ones. It is the right starting point and not the last word.

`locale` is accepted and forwarded, but it does not select the dictionary. ICU applies a single combined Chinese-Japanese word list to Han and kana runs, chosen by the script of the text, so Chinese and Japanese segment the same way under `"zh"`, `"ja"` or the session default.

`"character"` needs nothing at all: every CJK character becomes its own token and runs of non-CJK text are split on whitespace. Whitespace is never a token, the ideographic space U+3000 included,

even though has_cjk() counts it as CJK. It is character tokenisation rather than word segmentation, and for Chinese it will cut two-character words in half. It is a baseline, not an answer.

Value

cjk_segmenters() returns a character vector of engine names. register_cjk_segmenter() is called for its side effect and returns name invisibly.

The two engines do not tokenise punctuation alike

This matters when comparing counts, so it is worth stating rather than leaving to be discovered. "icu" drops punctuation and symbols along with whitespace; "character" keeps CJK punctuation as tokens, because those code points are in blocks cjk_blocks() lists, and keeps a run of non-CJK text whole up to the next space.

The upshot is that the same sentence yields different token counts, and the gap is punctuation rather than a disagreement about where words end. An emoji is dropped by "icu" and kept by "character" for the same reason. Filter or compare accordingly.

Zero-width characters divide the same way, in three groups. "character" keeps all of them wherever they sit. "icu" drops U+200B everywhere, because ICU classifies the zero-width space as "none" in every position and this engine asks for "none" to be skipped; it drops U+200D, U+00AD, U+2060, a combining mark and a variation selector only when one begins the string, which is the one position ICU calls "none" for those. A byte-order mark survives both engines, and for a different reason worth keeping straight: stringi removes a leading U+FEFF before ICU is called at all, so that one is data loss rather than policy, and both engines put it back.

Registering another segmenter

For Japanese, gibasa binds MeCab and is on CRAN; it gives part of speech and lemma, which "icu" does not.

jiebaR, which binds cppjieba, is the usual choice for Chinese. It was archived from CRAN on 2025-05-01, so it cannot be a dependency of a CRAN package and install.packages() will not find it; install it from source with remotes::install_github("qinwf/jiebaR"). Once you have it, four lines make it an engine:

```
register_cjk_segmenter("jiebar", function(x, ...) {
  worker <- jiebaR::worker(...)
  lapply(x, function(s) {
    if (is.na(s)) return(NA_character_)
    if (!nzchar(s)) return(character(0))
    as.character(jiebaR::segment(s, worker))
  })
})
```

The same shape works for any segmenter you can call from R.

The engine contract

An engine is any function taking `(x, ...)` – a character vector and the dots from `cjk_segment()` – and returning a list the same length as `x`, each element a character vector of tokens. `NA` input should give `NA_character_` and the empty string should give `character(0)`; `cjk_segment()` checks the shape and complains if an engine breaks the contract. A plain list is required: a data frame is a list too, but `length()` on one counts columns rather than elements, so it is refused rather than quietly mistaken for a list of tokens.

A leading byte-order mark is preserved by both built-in engines, and by neither of the stringi calls underneath them: `stri_split_charclass()` and `stri_split_boundaries()` each read a leading U+FEFF as a byte-order mark and drop it. Both engines count it off and put it back, so a column read from a BOM-carrying CSV tokenises without losing a character. An engine you register yourself gets no such treatment – if it calls stringi, it needs the same guard.

The elements are checked too, for the same reason one level down. An atomic element is coerced with `as.character()`, so an engine returning integers or a factor works; a list element is refused, because `as.character()` deparses a list rather than coercing it and `list(c(1, 2))` would otherwise arrive as the single token `"c(1, 2)"` – R code spelled out as data.

Passing arguments to an engine

Anything in `...` goes to the engine, which is how you configure one. Name those arguments so that they are not a prefix of an argument of the verb itself: `...` comes after `engine` in `cjk_segment()`, and after `data` and `col` in `cjk_tokens()`, so R's partial matching claims a prefix of one of those before the dots ever see it.

It is worth knowing because the result does not look like an argument-matching problem. `cjk_tokens(df, text, "mine", c = 1)` matches `c` to `col`, which pushes the bare `text` into `engine`, where it resolves to `graphics::text()` – a function, so it is accepted as an engine – and the error you get is about plotting. Single letters and short prefixes are the risk: `c`, `co`, `d`, `da`, `e`, `en`, `eng`. A longer name, or a closure that captures the setting instead of passing it, avoids the question:

```
register_cjk_segmenter("mine", function(x, ...) my_segmenter(x, cutoff = 1))
```

What registering does, and does not, undo

A registration lasts for the rest of the session and there is no function to remove one. Registering the same name again replaces it, which is the way to correct an engine you got wrong.

A name that matches a built-in shadows it. That is deliberate – it is how you substitute your own tokeniser for `"character"` without this package getting a say – but it is worth knowing that `"character"` is a natural name for an engine and taking it hides the built-in for the session, with nothing in `cjk_segmenters()` to show that anything changed. Pick a distinct name unless shadowing is what you meant.

See Also

`cjk_segment()`, `cjk_tokens()`.

Examples

```
cjk_segmenters()

# an engine that splits on an explicit marker
register_cjk_segmenter("pipe", function(x, ...) strsplit(x, "|",
                                                          fixed = TRUE))
cjk_segment("\u4e2d\u6587|\u5f88\u597d", engine = "pipe")
```

`cjk_sentences` *Split text into sentences*

Description

Splits each string into sentences using ICU's sentence break iterator, which knows the CJK terminators – the ideographic full stop U+3002, the fullwidth exclamation mark U+FF01 and the fullwidth question mark U+FF1F – as well as the ASCII ones.

Usage

```
cjk_sentences(x, locale = NULL)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`locale`	ICU locale for the break iterator, e.g. `"ja"`. `NULL`, the default, uses the session default. See Details – it does not change where CJK sentences break.

Details

A regular expression on `[.!?]` finds no boundary at all in Chinese or Japanese text, because neither uses those characters to end a sentence. This is the reason the function exists: it is the sentence-level counterpart of `cjk_segment()`, and the usual first step before sentence-level classification, translation or embedding.

Boundaries follow Unicode Annex #29. `locale` is accepted and forwarded, but do not expect it to change where CJK sentences break: across Chinese, Japanese, Korean and mixed text it gave identical output under `"zh"`, `"ja"`, `"ko"`, `"en"` and the session default. ICU can tailor break rules per locale in principle, and a locale it has no break data for is an error here rather than a silent fallback, which is why the argument is worth having at all.

Value

A list the same length as `x`, each element a character vector of sentences. `NA` gives `NA_character_`; the empty string gives `character(0)`.

What comes back

The spans are ICU's, returned unmodified, which has one visible consequence: whitespace that sits between two sentences stays attached to the end of the first. Nothing is trimmed, because trimming would make this verb edit text rather than divide it. Call `trimws()` on the result if you want it gone.

A limitation worth knowing

Abbreviations are a hard case in Latin text and ICU's default rules do not carry an abbreviation list, so `"Dr. Smith went home."` splits after `"Dr. "`. CJK text does not have the problem, because the terminator is unambiguous. If your corpus is mixed and abbreviation-heavy, check the result before trusting it.

See Also

`cjk_segment()` for words, `cjk_ngrams()` for character n-grams.

Examples

```
# "I am happy today. And you?"
cjk_sentences("\u6211\u4eca\u5929\u5f88\u958b\u5fc3\u3002\u4f60\u5462\uff1f")

# ASCII terminators work too, and mixed text is fine
cjk_sentences("First one. \u4e2d\u6587\u4e5f\u53ef\u4ee5\u3002")

# One row per sentence, the way cjk_tokens() gives one row per token.
# There is no tidy verb for this because the list is already the hard
# part; rep() over lengths() is the whole of the rest.
docs <- data.frame(
  id = 1:2,
  text = c("\u6211\u5f88\u958b\u5fc3\u3002\u4f60\u5462\uff1f", "One. Two.")
)
sents <- cjk_sentences(docs$text)
data.frame(
  id = rep(docs$id, lengths(sents)),
  sentence = unlist(sents, use.names = FALSE)
)
```

`cjk_simplify` *Convert between simplified and traditional Han*

Description

`cjk_simplify()` maps traditional Han characters to simplified; `cjk_traditionalize()` maps the other way. Characters outside Han, and characters with no counterpart, are left alone.

Usage

```
cjk_simplify(x)

cjk_traditionalize(x)
```

Arguments

x	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

This is ICU's `Simplified-Traditional` pair, and it is context-aware rather than a per-character table. Simplified to traditional is genuinely one-to-many – U+53D1 is U+767C ("to send") or U+9AEE ("hair") depending on the word, and U+5E72 is U+4E7E, U+5E79 or U+5E72 – and ICU picks correctly from the surrounding characters. Twenty such words were checked, including the ones where the same character splits both ways ("after" against "empress", "inside" against "kilometre"), and every one came out right.

What it does not do is substitute regional vocabulary. The two standards differ in the words they use as well as in glyph shape, and that is lexical rather than orthographic. The simplified word for "software", U+8F6F U+4EF6, converts to U+8EDF U+4EF6 – the correct characters, and the wrong word in Taiwan, where the term is U+8EDF U+9AD4. "Mouse" becomes U+9F20 U+6A19 rather than U+6ED1 U+9F20.

So use these for script conversion, which is what they are good at, and reach for OpenCC when you need Taiwanese or Hong Kong idiom – its regional configurations are what carry that.

Value

A character vector the same length as `x`. `NA` gives `NA`.

See Also

`cjk_romanize()`, `to_halfwidth()` for the width axis.

Examples

```
# U+6F22 U+5B57 / U+6C49 U+5B57, "Han characters"
cjk_simplify("\u6f22\u5b57")        # traditional -> simplified
cjk_traditionalize("\u6c49\u5b57")  # and back

# context decides a one-to-many character: "hair" vs "to send"
cjk_traditionalize(c("\u5934\u53d1", "\u53d1\u9001"))
```

```
# but regional vocabulary is not substituted: the Taiwanese word differs
cjk_traditionalize("\u8f6f\u4ef6")  # -> U+8EDF U+4EF6, not U+8EDF U+9AD4
```

`cjk_sort` *Sort CJK text by pronunciation or stroke*

Description

`cjk_sort()` returns `x` ordered by an ICU collation; `cjk_order()` returns the permutation that would do it, for reordering a data frame.

Usage

```
cjk_sort(x, locale = NULL, decreasing = FALSE)

cjk_order(x, locale = NULL, decreasing = FALSE)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`locale`	ICU locale naming the collation, e.g. `"zh"`. `NULL`, the default, uses the session default – which is unlikely to order Han usefully, so name one.
`decreasing`	Reverse the order.

Details

`sort()` reads `LC_COLLATE`, so the order it gives Han text is a property of the session rather than of the data. Under `C` or `en_US.UTF-8` it falls back to code point order; under `zh_CN.utf8` glibc supplies a pinyin collation and the answer changes; under `ja_JP.utf8` it changes again. None of those is wrong in itself, and that is the difficulty: the same script sorts a name column differently on two machines and says nothing.

The fallback order is worth knowing, because it is not arbitrary. The unified ideographs are laid out in KangXi radical-stroke order, so within the base block sorting by code point sorts by radical. What it is not is phonetic, which is what someone sorting names is after. It also stops being radical order across blocks: every Extension A character (U+3400-U+4DBF) sorts ahead of every base-block one, so a rare character lands nowhere near the common characters sharing its radical.

ICU carries real collations. With `locale = "zh"` Han sorts by pinyin, so a column of Chinese surnames comes out in the order a Chinese reader expects.

Value

`cjk_sort()` a character vector the same length as `x`; `cjk_order()` an integer vector of indices.

Choosing the order

The collation is selected by the locale string, including the BCP 47 variants:

- `"zh"` – Mandarin pinyin.
- `"zh-u-co-stroke"` – stroke count, the convention in many indexes.
- `"zh-u-co-zhuyin"` – bopomofo order.
- `"ja"` and `"ko"` – Japanese and Korean.

A locale whose *language* ICU has no collation for is an error here rather than the silent fallback to root collation stringi would otherwise give you, because that fallback produces a plausible-looking wrong order. An unrecognised *region* is not an error: `"zh-CH"`, a typo for `"zh-CN"`, resolves to `"zh"` and still sorts by pinyin, which is the right answer. The language is matched without regard to case, as BCP 47 specifies, so `"ZH"` is `"zh"`; and `"root"` asks for ICU's root collation by name, which is a choice rather than a fallback.

Missing values

`NA` sorts last and is kept, so the result is always the same length as `x`. `stringi::stri_sort()` drops missing values by default, which would silently shorten a column.

See Also

`cjk_romanize()`, whose `ascii = TRUE` output gives a sort key you can use where no ICU locale is available.

Examples

```
# Chinese surnames: Zhang, Wang, Li
x <- c("\u5f35", "\u738b", "\u674e")

sort(x)                        # depends on LC_COLLATE
cjk_sort(x, locale = "zh")     # pinyin: Li, Wang, Zhang -- on any machine

# stroke order instead
cjk_sort(x, locale = "zh-u-co-stroke")

# reorder a data frame
df <- data.frame(name = x)
df[cjk_order(x, locale = "zh"), , drop = FALSE]
```

cjk_strip_punct *Remove punctuation from CJK text*

Description

`cjk_strip_punct()` removes punctuation using Unicode's own category rather than a POSIX class, so the same call removes the same characters on every machine. It is the step before `cjk_ngrams()` or `cjk_segment()` that stops punctuation becoming part of a token.

Usage

```
cjk_strip_punct(x, replacement = " ", symbols = FALSE)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`replacement`	A single string to put in place of each removed character. Defaults to `" "`; see above for why it is not `""`.
`symbols`	Also remove General_Category `S`: the fullwidth tilde U+FF5E, currency signs including U+FFE5, and mathematical operators. Defaults to `FALSE`, because a currency sign is often content.

Details

The usual spelling, `gsub("[[:punct:]]", "", x)`, is unreliable on CJK in two separate ways. R's default engine resolves `[:punct:]` through the C library, so under `LC_ALL=C` it removes no CJK punctuation and under a UTF-8 locale it removes all of it – the same script gives two answers on two machines. Passing `perl = TRUE` does not fix it but hides it: PCRE's POSIX classes are ASCII-only unless `(*UCP)` is set, so that spelling silently removes nothing from Chinese or Japanese in any locale at all.

This removes Unicode General_Category `P`, which is the ideographic full stop U+3002, the ideographic comma U+3001, the fullwidth comma, question mark and exclamation mark, the corner and fullwidth brackets, and the katakana middle dot U+30FB that separates the parts of a transliterated name – along with ASCII punctuation, so mixed text needs only one pass.

Value

A character vector the same length as `x`. `NA` gives `NA`, and a leading byte-order mark is preserved.

What it deliberately keeps

U+30FC, the katakana-hiragana prolonged sound mark, is a *modifier letter* and not punctuation. It looks like a dash and is not one: it carries the long vowel in most Japanese loanwords, so removing it turns the words for coffee and ramen into something else. Testing the category keeps it; any rule phrased about dashes does not.

The ideographic space U+3000 is whitespace rather than punctuation and is left alone too. `cjk_segment()` and `cjk_ngrams()` already ignore whitespace, so there is nothing to do about it here.

Why the default replaces rather than deletes

`replacement` defaults to a space, not `""`. Deleting a full stop closes the gap it left, and the two characters that were on either side of it become adjacent: `cjk_ngrams()` then reports a bigram spanning a sentence boundary, a "word" that was never written. A space keeps the boundary, and every verb here that walks a string already declines to cross whitespace. Pass `replacement = ""` when you want the characters closed up – for a display string rather than for tokenising.

See Also

`cjk_ngrams()` and `cjk_segment()`, the verbs this feeds; `cjk_normalize()` for folding width and compatibility variants.

Examples

```
x <- "\u4ed6\u8aaa\uff08\u4eca\u5929\uff09\u3002\u771f\u597d"
cjk_strip_punct(x)
cjk_strip_punct(x, replacement = "")

# the prolonged sound mark is a letter, not a dash, and survives
cjk_strip_punct("\u30b3\u30fc\u30d2\u30fc\u3001\u30e9\u30fc\u30e1\u30f3")

# a space keeps n-grams from spanning the full stop
cjk_ngrams(cjk_strip_punct("\u597d\u3002\u5929"))
cjk_ngrams(cjk_strip_punct("\u597d\u3002\u5929", replacement = ""))
```

`cjk_summary` *Summarise CJK content in a text column*

Description

`cjk_summary()` reports how much of a text column is CJK: how many entries contain any CJK at all, what share of entries that is, and the mean share of each entry's characters that are CJK.

Usage

```
cjk_summary(data, col)
```

Arguments

`data`	A data frame or tibble containing a text column.
`col`	The text column to scan, supplied unquoted. A non-character column is coerced with `as.character()`; see `has_cjk()` for why that makes a numeric column a poor thing to measure. A list-column is an error rather than a coercion, for the reason given there.

Details

`prop_with_cjk` counts an entry once however much CJK it holds, so it answers "how many of these documents are CJK at all". `mean_ratio` averages `cjk_ratio()` over the entries that have one, so it answers "how CJK are they". A corpus of English with one ideograph per row scores high on the first and near zero on the second.

`NA` entries never count as containing CJK, and they do count towards `n_docs` – so they are in the denominator of `prop_with_cjk` and dilute it: a column that is half missing cannot score above 0.5. `mean_ratio` is the one figure they are dropped from, along with empty strings, because neither has a ratio to contribute.

Grouping is ignored: the result is always one row for the whole column. Use `dplyr::group_modify()` if you need it per group.

Value

A one-row tibble with columns `n_docs` (all entries), `n_with_cjk` (entries holding at least one CJK character), `prop_with_cjk` and `mean_ratio`. `prop_with_cjk` is `NA` for a zero-row column, and `mean_ratio` is `NA` when no entry has a ratio to contribute.

See Also

`cjk_char_counts()` for the per-character breakdown, `cjk_ratio()` for the per-row measure this averages.

Examples

```
# U+4E2D U+6587 is "Chinese writing": all CJK, then mixed, then neither
df <- data.frame(
  text = c("\u4e2d\u6587", "mixed \u4e2d\u6587 text", "plain ASCII", NA)
)
cjk_summary(df, text)
```

`cjk_tokens` *One row per token*

Description

`cjk_tokens()` segments a text column and returns one row per token, carrying the other columns along. It is the CJK-aware counterpart of tidytext's `unnest_tokens()`, which splits on whitespace and therefore returns CJK sentences whole.

Usage

```
cjk_tokens(data, col, engine, ..., output = "token")
```

Arguments

data	A data frame or tibble containing a text column.
col	The text column to scan, supplied unquoted. A non-character column is coerced with `as.character()`; see `has_cjk()` for why that makes a numeric column a poor thing to measure. A list-column is an error rather than a coercion, for the reason given there.
engine	Name of a segmentation engine, or a function implementing one. Required; see `cjk_segmenters()`.
...	Passed to the engine. Name these so they are not a prefix of `engine` (or of `data`/`col` in `cjk_tokens()`); see "Passing arguments to an engine" in `cjk_segmenters()`.
output	Name for the token column. Defaults to `"token"`. It follows `...`, so it has to be given by its full name.

Details

Rows that produce no tokens – empty strings, and text with nothing an engine recognises – are dropped, as they are in **tidytext**. `NA` text yields one row with an `NA` token, so a missing document does not silently vanish from the output.

The token column is called `token` and is added to `data`; an existing column of that name is replaced. Set `output` to put the tokens somewhere else when `data` already has a column worth keeping under that name – the same escape hatch **tidytext**'s `unnest_tokens()` gives you. It comes after `...` and so has to be named in full, which is what stops it competing with an engine argument for a prefix. As with `cjk_segment()`, `engine` is required.

Grouping is dropped, as it is by `cjk_summary()`: the result is a plain tibble even when `data` is a `grouped_df`. Regroup it afterwards if you need the groups back.

Value

`data`, as a tibble, with one row per token and an added `token` column. Row order follows the input, and tokens within a row follow the text.

See Also

`cjk_segment()` for the vector version, `cjk_char_counts()` when you want characters rather than words.

Examples

```
# "I am very happy", then a row mixing Latin with "Chinese writing"
df <- data.frame(
  id = 1:2,
  text = c("\u6211\u5f88\u958b\u5fc3", "hello \u4e2d\u6587")
)
cjk_tokens(df, text, engine = "character")
```

`cjk_truncate` *Truncate text to a display width*

Description

`cjk_truncate()` shortens each string so that it fits in `width` terminal columns, appending an ellipsis when anything was removed. Because CJK characters are two columns wide, truncating by character count overshoots the available space by up to a factor of two.

Usage

```
cjk_truncate(x, width, ellipsis = "...")
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`width`	Maximum display width in columns. Recycled against `x`; a pair of lengths that does not recycle cleanly is an error.
`ellipsis`	String to append when the text was shortened. Defaults to `"..."`. The single-character ellipsis U+2026 is one column rather than three, so more of the text survives.

Details

The result is never wider than `width`. When a string has to be shortened, the ellipsis is included in the budget, so the kept text is trimmed to `width - cjk_width(ellipsis)` columns. If `width` is too small even for the ellipsis, the ellipsis itself is truncated.

Cuts never separate a combining mark from the character it modifies: a zero-width character immediately after the cut point is carried along with it.

Value

A character vector the same length as the recycled inputs. Strings that already fit are returned unchanged. `NA` input, and an `NA` width, give `NA`.

See Also

`cjk_pad()` for the other direction; `cjk_width()` for the measure both use.

Examples

```
# six columns would be three ideographs, but "..." is counted against
# the budget and costs three of them, so one ideograph is what fits
cjk_truncate("\u4e2d\u6587\u4e2d\u6587\u4e2d\u6587", 6)

# the same six columns hold three ASCII characters plus the ellipsis
cjk_truncate("abcdefghij", 6)

# drop the ellipsis and all six columns are available again
cjk_truncate("\u4e2d\u6587\u4e2d\u6587\u4e2d\u6587", 6, ellipsis = "")

# already fits, so nothing happens
cjk_truncate("\u4e2d\u6587", 10)
```

`cjk_width` *Display width in terminal columns*

Description

`cjk_width()` returns the number of columns each string occupies in a monospaced terminal. CJK characters occupy two columns, not one, which is the reason `nchar()` misaligns every console table containing CJK text.

Usage

```
cjk_width(x)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

Width follows Unicode Annex #11 (East Asian Width). Characters whose East Asian Width is Wide or Fullwidth are two columns; combining marks and format characters (general categories `Mn`, `Me`, `Cf`), C0 and C1 control codes, and Hangul Jamo medial vowels and final consonants are zero; the rest are one. Treat that as the shape of the answer rather than the whole of it: recent ICU also gives two columns to several thousand symbols and pictographs that Annex #11 itself calls neutral or ambiguous. Where a layout turns on one particular character, measure it rather than deriving it from this list.

The computation is `stringi::stri_width()`, which reads the Unicode tables shipped with ICU, the Unicode Consortium's C library. `cjk_width()` exists so that the width, the padding and the truncation in a CJK pipeline all read the same way; if width is all you need, `stri_width()` is the more direct call.

East Asian Ambiguous characters render as two columns in a CJK-configured terminal and one everywhere else, and no library can resolve that without being told which terminal it is writing to. `cjk_width()` reports whatever the ICU build behind your **stringi** decided, and that answer has moved: ICU once called the whole class one column, and now gives two to several hundred of them, the box-drawing characters and the degree sign among them. Greek and Cyrillic letters have stayed at one throughout.

So which side a given ambiguous character falls on is a property of the **stringi** build in front of you, not of this package, and not something this page can usefully enumerate. Measure it with `cjk_width()` if it matters, and keep ambiguous-width characters out of any table that has to line up on someone else's machine.

Value

An integer vector the same length as `x`. `NA` input gives `NA`; the empty string gives `0`.

See Also

`cjk_pad()` and `cjk_truncate()`, which lay text out by width; `stringi::stri_width()` for the underlying computation.

Examples

```
# two characters, four columns
cjk_width("\u4e2d\u6587")

# two characters, two columns
cjk_width("ab")

# nchar() cannot tell these apart; cjk_width() can
nchar(c("\u4e2d\u6587", "abcd"))
cjk_width(c("\u4e2d\u6587", "abcd"))
```

`cjk_wrap` *Wrap text to a display width*

Description

Breaks each string into lines no wider than `width` terminal columns, completing the layout set with `cjk_pad()` and `cjk_truncate()`. Like them, it counts columns rather than characters.

Usage

```
cjk_wrap(x, width, indent = 0L, exdent = 0L, locale = NULL)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`width`	Target width in columns. Recycled against `x`.
`indent`	Columns to indent the first line of each string by.
`exdent`	Columns to indent every line after the first by.
`locale`	ICU locale selecting the line-breaking style, e.g. `"ja"` or `"ja@lb=strict"`. `NULL`, the default, uses the session default – which means the result depends on where it is run; see Details.

Details

The break positions come from ICU's implementation of Unicode Annex #14, the line breaking algorithm, which is what makes this more than `strwrap()` with a different counter. CJK text is mostly breakable *between* characters – there are no spaces to break at – but not everywhere: a closing bracket may not begin a line, an opening bracket may not end one, and the ideographic full stop U+3002 may not start one either. Splitting every `width` columns would put breaks in all three places. Latin runs inside the same string still break on spaces. A fourth rule, about small kana, holds only in some locales – see below.

Lines are returned joined by `\n`, so the result is the same length as `x` and can go straight to `cat()`. `strsplit(out, "\n", fixed = TRUE)` gives the lines separately.

A line comes out wider than `width` only where no break is allowed inside it, and then ICU emits it whole rather than looping. That happens in three cases: a character wider than the budget (a CJK character needs two columns, so `width = 1` cannot be honoured), a run with no break opportunity in it, such as a long Latin word or a stretch of a URL, and an `indent` or `exdent` that leaves no room for the text after it.

Value

A character vector the same length as the recycled inputs, each element the wrapped text with lines separated by `\n`. `NA` gives `NA`. As in `cjk_pad()` and `cjk_truncate()`, a `width` longer than `x` recycles `x` up to it rather than being an error.

The break style depends on the locale

Annex #14 defines three line-breaking styles – strict, normal and loose – and which one applies is tailored per locale. The difference that shows up in CJK text is small kana: under `"en"` a small kana may not begin a line, and under `"ja"`, which uses the looser style, it may. So the same call wraps Japanese differently depending on the session locale, which is why `locale` is an argument here rather than left implicit. Name it when the output has to be reproducible, and `"ja@lb=strict"` if you want the strict style for Japanese.

The rules that hold whatever the locale are the ones about punctuation: a closing bracket or an ideographic full stop never begins a line, and an opening bracket never ends one.

Lines are filled greedily

Each line takes as much text as fits before the next break opportunity would overflow it, which is what a terminal does. `stri_wrap()`'s own default is an optimal fit that evens out line lengths across a paragraph; it is not used here, because its cost grows far faster than the text (40,000 characters of CJK exhaust 1.5 GB of memory), and because the breaks it picks for the start of a paragraph depend on how the paragraph ends, so appending a sentence can re-flow every line above it. Greedy breaks depend only on the text up to them.

A leading byte-order mark survives, which takes work: stringi drops one. A U+FEFF *elsewhere* in the string may not, because re-flowing can put it at the start of a segment, where stringi reads it as a byte-order mark again and removes it. Unicode deprecated U+FEFF for any use other than marking the start of a stream, so this only bites text that is already using it against the standard's advice – but it is a content change, and this page would rather say so than have you find it.

Three behaviours differ from `cjk_pad()` and `cjk_truncate()`, because this verb re-flows text rather than measuring it in place. Existing newlines in `x` are whitespace to the algorithm and are replaced by the new line breaks, so `"a\nb"` wrapped wide comes back as `"a b"`; wrap the pieces separately if the original breaks are meaningful. A string of nothing but whitespace re-flows to `""`, where `cjk_pad()` would have kept it.

And the output is in normalisation form C. ICU's line-breaking works on normalised text, so `stri_wrap()` normalises, and 1,120 code points therefore come back changed. 1,002 of them are the CJK Compatibility Ideographs (U+F900-U+FAD9 and U+2F800-U+2FA1D): `cjk_wrap("\uf900", 2)` is U+8C48. The other 118 are the Hebrew presentation forms, a handful of musical symbols, and 71 scattered singletons and composition exclusions: the Kelvin, Ohm and Angstrom signs, Greek letters with an oxia, Devanagari letters with a nukta, and U+2329 and U+232A, which become the CJK angle brackets U+3008 and U+3009. These are exactly the code points NFC changes, no more and no fewer, each a duplicate of a character or sequence Unicode prefers and kept for compatibility. Everyday text, precomposed Latin and Hangul syllables included, is already NFC and passes through untouched. `cjk_pad()`, `cjk_truncate()` and `cjk_segment()` do **not** normalise, so this is the one layout verb that can change a character. Undoing it would mean re-aligning re-flowed output against the input, which risks corrupting ordinary text to protect a deprecated corner of it; running `cjk_normalize()` first is the way to make the normalisation explicit and expected.

See Also

`cjk_pad()` and `cjk_truncate()` for the fixed-width forms, `cjk_width()` for the measurement itself.

Examples

```
# "I am happy today, because the weather is very good"
x <- paste0("\u6211\u4eca\u5929\u5f88\u958b\u5fc3\uff0c",
            "\u56e0\u70ba\u5929\u6c23\u975e\u5e38\u597d")
cat(cjk_wrap(x, 12), "\n")

# every line is within the budget, measured in columns
cjk_width(strsplit(cjk_wrap(x, 12), "\n", fixed = TRUE)[[1]])
```

```
# hanging indent
cat(cjk_wrap(x, 12, exdent = 2), "\n")

# the strict style, so a small kana never begins a line
cat(cjk_wrap("\u304d\u3087\u3046\u306f\u3068\u3066\u3082\u3044\u3044",
             6, locale = "ja@lb=strict"), "\n")
```

---

has_cjk *Does the text contain CJK characters?*

Description

has_cjk() reports, for each element, whether the string contains at least one character from a CJK Unicode block.

Usage

```
has_cjk(x)
```

Arguments

x	A character vector. Anything else is coerced with as.character(). That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with options(scipen) and options(OutDec), and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – as.character() deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

"CJK" here means any block listed by cjk_blocks(), which includes CJK punctuation and the halfwidth and fullwidth forms as well as the ideographs and the phonetic scripts. That is deliberate – a column typed with a CJK input method carries the punctuation too – but it does mean that a string of nothing but ideographic full stops (U+3002) is TRUE. Use cjk_script() when you need to know *which* kind of CJK you have.

Value

A logical vector the same length as x. NA input gives NA; the empty string gives FALSE.

See Also

cjk_ratio() for how much of the text is CJK, cjk_script() for which script it is.

Examples

```
# U+4E2D U+6587, "Chinese writing"
has_cjk(c("\u4e2d\u6587", "plain ASCII", NA))

# the ideographic full stop U+3002 counts, by design
has_cjk("\u3002")
```

`to_halfwidth` *Normalise fullwidth and halfwidth forms*

Description

`to_halfwidth()` narrows fullwidth ASCII to ASCII, and `to_fullwidth()` widens ASCII to fullwidth. Both widen halfwidth katakana to its fullwidth form. Nothing else in the string is touched.

Usage

```
to_halfwidth(x, compose = TRUE)

to_fullwidth(x)
```

Arguments

`x`	A character vector. Anything else is coerced with `as.character()`. That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with `options(scipen)` and `options(OutDec)`, and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – `as.character()` deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.
`compose`	Fold a syllable and a following voiced mark into the single precomposed code point. Defaults to `TRUE`. `FALSE` leaves the pair as two code points, which is occasionally what you want if you are counting marks rather than characters.

Value

A character vector the same length as `x`. `NA` input gives `NA`; the empty string gives the empty string.

Why not NFKC

`NFKC` normalisation does fix character width, and it is what most advice recommends. It also rewrites ligatures, superscripts and subscripts, Roman numerals, circled and parenthesised numbers, the no-break space, and the CJK compatibility ideographs. A user who wants fullwidth digits narrowed before parsing them as numbers almost never wants the rest of that, and the damage is silent. These functions change width and nothing else.

What is mapped

- Fullwidth ASCII U+FF01-U+FF5E and ASCII U+0021-U+007E, which differ by a constant offset of `0xFEE0`.
- The ideographic space U+3000 and the ASCII space U+0020.
- Halfwidth katakana U+FF61-U+FF9F, which always maps *to* the fullwidth form – in both directions. This is the ordinary Japanese convention (alphanumerics halfwidth, katakana fullwidth), and it is forced: the voiced syllables have no halfwidth form of their own, so fullwidth is the only representation that survives a round trip.

That is the whole of it, and the rest of the Halfwidth and Fullwidth Forms block is left alone – which is worth naming, because those code points sit immediately beside the ones above. The fullwidth currency and sign forms U+FFE0-U+FFE6 (cent, pound, not, macron, broken bar, yen, won) keep their width, so `to_halfwidth()` narrows the digits of a price and leaves the currency symbol fullwidth. So do the halfwidth Hangul jamo U+FFA0-U+FFDC, the halfwidth symbol forms U+FFE8-U+FFEE, and the fullwidth white parentheses U+FF5F and U+FF60. None of them is ASCII on either side, and fullwidth ASCII is what these functions promise; `NFKC` maps all of them, along with everything else named under "Why not NFKC" above.

Voiced marks

Halfwidth katakana writes a voiced syllable as two code points, a bare syllable followed by a voiced sound mark. Mapping those to fullwidth one-for-one leaves the pair intact, so the text still has two code points where a reader sees one character, and it will not match a literal written the normal way.

With `compose = TRUE`, the default, the pair is folded into the single precomposed code point: U+FF76 U+FF9E becomes U+30AC, one character, rather than U+30AB followed by U+309B. Voicing adds one to the base throughout the ka, sa, ta and ha rows, and to the katakana iteration mark U+30FD; the semi-voiced mark adds two and applies to the ha row only. Five characters break the arithmetic and are mapped explicitly: U+30A6 voices to U+30F4, and the wa-row characters U+30EF, U+30F0, U+30F1 and U+30F2 voice into U+30F7 to U+30FA. Every pair agrees with Unicode NFC composition.

Composition applies to katakana, which is what the width mapping produces. Both the spacing marks (U+309B, U+309C) and the combining marks (U+3099, U+309A) are recognised, so katakana that arrived already decomposed is composed too.

`to_fullwidth()` always composes, because a fullwidth string carrying an uncomposed voiced mark is not a form anyone wants.

One deliberate difference from NFKC and from ICU

The Unicode compatibility decomposition of U+FF9E is the *combining* mark U+3099, so `NFKC` maps the halfwidth voiced mark onto a combining character, and so does ICU's `Halfwidth-Fullwidth` transform. These functions map it to the *spacing* mark U+309B instead, and U+FF9F to U+309C.

The difference is only visible with `compose = FALSE`, and the spacing mark is the safer of the two there: a combining mark left loose attaches itself to whatever character happens to precede it. ICU shows the hazard on its own transform – `"a"` followed by U+FF9E comes back as U+FF41 U+3099, a fullwidth `a` wearing a voiced sound mark. With `compose = TRUE`, the default, the question does not arise: the mark is folded into the syllable and no bare mark survives either way.

See Also

cjk_normalize() for the Unicode normalisation forms, which fold width along with much else, cjk_width() for measuring the result.

Examples

```
# fullwidth digits will not parse as numbers until they are narrowed
to_halfwidth("\uff11\uff12\uff13")
as.numeric(to_halfwidth("\uff11\uff12\uff13"))

# halfwidth katakana is widened, and the voiced mark is composed:
# U+FF76 U+FF9E (two code points) becomes U+30AC (one)
to_halfwidth("\uff76\uff9e")
nchar(to_halfwidth("\uff76\uff9e"))
nchar(to_halfwidth("\uff76\uff9e", compose = FALSE))

# ASCII round-trips exactly, in both directions
to_halfwidth(to_fullwidth("abc 123"))
to_fullwidth("abc")
```

---

to_hiragana *Convert between hiragana and katakana*

Description

to_hiragana() maps katakana to hiragana and to_katakana() maps the other way. Kanji, Latin text and punctuation are untouched.

Usage

```
to_hiragana(x)

to_katakana(x)
```

Arguments

x	A character vector. Anything else is coerced with as.character(). That coercion is R's, not this package's, so a numeric vector is measured as R chooses to write it – which moves with options(scipen) and options(OutDec), and can therefore differ between sessions. Convert deliberately if you mean to measure numbers; these verbs are for text. A list is *not* coerced – as.character() deparses one rather than coercing it, so the text measured would be the R code that builds the list – so a list, a data frame or a function is an error naming what to do instead.

Details

The two kana syllabaries encode the same sounds, so this conversion is unambiguous – unlike `cjk_romanize()` or `cjk_simplify()`, there is no context that could change the answer. It is the normalisation you want before comparing or grouping Japanese text, where the same word may be written either way for emphasis.

It is a normalisation, not a reversible mapping. Applied to text holding both syllabaries it erases the distinction between them, and that distinction means something: katakana marks loanwords, onomatopoeia and emphasis. `to_katakana(to_hiragana(x))` turns any hiragana in `x` into katakana, and it is not exact even on text that was all katakana, because ICU's mapping makes a few choices of its own: the small katakana U+30F5 and U+30F6 become the full-size hiragana U+304B and U+3051, and the digraphs U+30FF (*koto*) and U+309F (*yori*) are spelled out as two kana each. Run it one way, and keep the original if you need to go back.

Halfwidth katakana is handled too, and composed while it is: the halfwidth voiced KA is two code points, U+FF76 and U+FF9E, and `to_katakana()` returns the single character U+30AC. So no width conversion is needed first, though `to_halfwidth()` and `to_fullwidth()` remain the way to move along the width axis without touching the syllabary.

Value

A character vector the same length as `x`. `NA` gives `NA`.

See Also

`to_halfwidth()` for the width axis, `cjk_script()` to see which kana a string is in.

Examples

```
to_hiragana("\u30ab\u30bf\u30ab\u30ca")   # katakana -> hiragana
to_katakana("\u3072\u3089\u304c\u306a")   # and back

# kanji is left alone
to_katakana("\u65e5\u672c\u8a9e\u3067\u3059")
```

Index

as.character(), *4*, *6*, *7*, *9*, *10*, *13–15*, *17*, *18*, *21*, *22*, *24*, *25*, *27*, *29–32*, *34*, *36*, *37*, *39*

cjk_blocks, 3
cjk_blocks(), *5*, *8*, *17*, *20*, *36*
cjk_char_counts, 4
cjk_char_counts(), *3*, *4*, *17*, *29*, *30*
cjk_compose_jamo *(*cjk_jamo*)*, 7
cjk_detect_language, 5
cjk_detect_language(), *4*
cjk_jamo, 7
cjk_ngrams, 9
cjk_ngrams(), *23*, *27*, *28*
cjk_normalize, 10
cjk_normalize(), *8*, *28*, *35*, *39*
cjk_order *(*cjk_sort*)*, 25
cjk_pad, 12
cjk_pad(), *31*, *33–35*
cjk_ratio, 14
cjk_ratio(), *4*, *9*, *29*, *36*
cjk_romanize, 15
cjk_romanize(), *24*, *26*, *40*
cjk_script, 17
cjk_script(), *4*, *6*, *8*, *36*, *40*
cjk_segment, 18
cjk_segment(), *10*, *16*, *19*, *21–23*, *27*, *28*, *30*, *35*
cjk_segmenters, 19
cjk_segmenters(), *18*, *21*, *30*
cjk_sentences, 22
cjk_sentences(), *10*
cjk_simplify, 23
cjk_simplify(), *16*, *40*
cjk_sort, 25
cjk_sort(), *12*
cjk_strip_punct, 27
cjk_summary, 28
cjk_summary(), *5*, *14*, *30*
cjk_tokens, 29
cjk_tokens(), *18*, *21*, *30*
cjk_traditionalize *(*cjk_simplify*)*, 23
cjk_truncate, 31
cjk_truncate(), *13*, *33–35*
cjk_width, 32
cjk_width(), *8*, *31*, *35*, *39*
cjk_wrap, 33

dplyr::group_modify(), *5*, *29*

findInterval(), *3*

graphics::text(), *21*

has_cjk, 36
has_cjk(), *3*, *4*, *6*, *14*, *20*, *29*, *30*

register_cjk_segmenter *(*cjk_segmenters*)*, 19

stringi::stri_pad(), *13*
stringi::stri_width(), *33*

to_fullwidth *(*to_halfwidth*)*, 37
to_fullwidth(), *11*, *12*, *40*
to_halfwidth, 37
to_halfwidth(), *11*, *12*, *24*, *40*
to_hiragana, 39
to_katakana *(*to_hiragana*)*, 39