Package 'tidycjk'

September 9, 2026

Type Package

Title Tidy Tools for Chinese, Japanese and Korean Text

Version 0.1.0

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 therefore a pluggable engine that the caller names explicitly rather than a bundled dictionary, 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 and truncates 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 -- without the collateral damage of a full 'NFKC' pass. 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/tidyckj/, https://github.com/PursuitOfDataScience/tidyckj

BugReports https://github.com/PursuitOfDataScience/tidyckj/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

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Youzhi Yu [aut, cre]

Maintainer Youzhi Yu <yuyouzhi666@icloud.com>

Repository CRAN

Date/Publication 2026-09-09 10:20:02 UTC

Contents

cjk_blocks 2
cjk_char_counts 4
cjk_detect_language 5
cjk_pad 6
cjk_ratio 7
cjk_script 8
cjk_segment 9
cjk_segmenters 10
cjk_summary 12
cjk_tokens 13
cjk_truncate 14
cjk_width 16
has_cjk 17
to_halfwidth 18

Index **21**

`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.

Two of the rows deserve comment. 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. And the halfwidth Hangul jamo at U+FFA0-U+FFDC are reported as `"fullwidth"` rather than `"hangul"`, because this table follows the block boundaries rather than the Unicode Script property; use `cjk_char_counts()` if you need to see exactly which code points a text contains.

All ten blocks of unified ideographs are covered – the base block and Extensions A through I. 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.

Ten is every block there was as of Unicode 16.0, which is what this table is current to. Unicode 17.0 added Extension J at U+323B0-U+3347F, and it is not here, so `has_cjk()` answers `FALSE` for an Extension J ideograph. That is a known limit of this version of the table rather than a judgement about the block, and it is the same gap the table once had at Extensions G, H and I.

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, so Bopomofo Extended (U+31A0-U+31BF) is here alongside Bopomofo. 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 parenthesised, circled and squared compatibility symbols in Enclosed CJK Letters and Months and CJK Compatibility. 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.

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.

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.

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

```
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.
`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.
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_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.

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.

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

```
cjk_ratio(c("\u4e2d\u6587", "half \u4e2d\u6587", "none", "", NA))
```

`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.

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.
`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

engine is required and has no default. The only engine **tidycjk** can ship without a dictionary is "character", which tokenises by character rather than by word – a different answer from the one you are asking for, and quietly returning it would be the mistake this package exists to avoid. cjk_segmenters() lists what is available and shows how to register a real word segmenter.

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

```
# 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 bundles no word segmenter and dispatches on a name instead.

One engine ships with the package. `"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.

Registering a word segmenter

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.

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_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.

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

```
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, ...)
```

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.
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

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. 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

```
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.
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 is three ideographs
cjk_truncate("\u4e2d\u6587\u4e2d\u6587\u4e2d\u6587", 6)

# ASCII, same budget
cjk_truncate("abcdefghij", 6)

# 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.

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"))
```

---

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.

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.
`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_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")
```

Index

as.character(), *4–6*, *8*, *9*, *12*, *14–18*

cjk_blocks, 2
cjk_blocks(), *4*, *9*, *17*
cjk_char_counts, 4
cjk_char_counts(), *3*, *9*, *13*, *14*
cjk_detect_language, 5
cjk_detect_language(), *3*
cjk_pad, 6
cjk_pad(), *15*, *17*
cjk_ratio, 7
cjk_ratio(), *3*, *13*, *18*
cjk_script, 8
cjk_script(), *3*, *6*, *17*, *18*
cjk_segment, 9
cjk_segment(), *10–12*, *14*
cjk_segmenters, 10
cjk_segmenters(), *9*, *10*, *12*, *14*
cjk_summary, 12
cjk_summary(), *4*, *8*, *14*
cjk_tokens, 13
cjk_tokens(), *9–12*, *14*
cjk_truncate, 14
cjk_truncate(), *7*, *17*
cjk_width, 16
cjk_width(), *15*, *20*

dplyr::group_modify(), *13*

findInterval(), *3*

graphics::text(), *12*

has_cjk, 17
has_cjk(), *3*, *4*, *6*, *8*, *11*, *12*, *14*

register_cjk_segmenter
        *(*cjk_segmenters*)*, 10

stringi::stri_pad(), *7*
stringi::stri_width(), *16*, *17*

to_fullwidth *(*to_halfwidth*)*, 18
to_halfwidth, 18