

stringi: Fast and Portable Character String Processing in R

Marek Gagolewski
Deakin University, Australia

Abstract

Effective processing of character strings is required at various stages of data analysis pipelines: from data cleansing and preparation, through information extraction, to report generation. Pattern searching, string collation and sorting, normalisation, transliteration, and formatting are ubiquitous in text mining, natural language processing, and bioinformatics. This paper discusses and demonstrates how and why **stringi**, a mature R package for fast and portable handling of string data based on the **ICU** library (International Components for Unicode), should be included in each statistician's or data scientist's repertoire to complement their numerical computing and data wrangling skills.

Keywords: **stringi**, character strings, text, **ICU**, Unicode, regular expressions, data cleansing, natural language processing, R.

This is a draft version of a paper on **stringi**, last updated on 2 March 2021.

Please cite as: Gagolewski M (2021). *stringi: Fast and Portable Character String Processing in R*. URL <https://stringi.gagolewski.com>.

1. Introduction

Stringology (Crochemore and Rytter 2003) deals with algorithms and data structures for processing character strings (Jurafsky and Martin 2008; Szpankowski 2001). From the perspective of applied statistics and data science, it is worth stressing that many interesting data sets first come in an unstructured or contaminated form, for instance when they have been fetched from different APIs or gathered by means of web scraping techniques.

Diverse data cleansing and preparation operations (Dasu and Johnson 2003; van der Loo and de Jonge 2018; see also Section 2 below for a real-world example) need to be applied before an analyst can begin to enjoy an orderly and meaningful data frame, matrix, or spreadsheet being finally at their disposal. Activities related to information retrieval, computer vision, bioinformatics, natural language processing, or even musicology can also benefit from including them in data processing pipeline (Jurafsky and Martin 2008; Kurtz *et al.* 2004).

Although statisticians and data analysts are usually very proficient in numerical computing and data wrangling, the awareness of how crucial text operations are in the overall data-oriented skill-set is yet to reach a more operational level. This paper aims to fill this gap.

Most statistical computing ecosystems provide only a basic set of text operations. In particular, base R (R Development Core Team 2021) is mostly restricted to pattern matching,

string concatenation, substring extraction, trimming, padding, wrapping, simple character case conversion, and string collation, see (Chambers 2008, Chapter 8) and Table 5 below. The **stringr** package (Wickham 2010), first released in November 2009, implemented an alternative, “tidy” API for text data processing (the list of 21 functions that were available in **stringr** at that time is given in Table 5). The early **stringr** featured a few wrappers around a subset of its base R counterparts. The latter, however – up to this day – not only is of limited scope, but also suffers from a number of portability issues; it may happen that the same code can yield different results on different operating systems; see Section 9 for some examples.

In order to significantly broaden the array of string processing operations and assure that they are portable, in 2013 the current author developed the open source **stringi** package (pronounced “stringy”, IPA [stringi]). Its API was compatible with that of early **stringr**’s, which some users found convenient. However, for the processing of text in different locales, which are plentiful, **stringi** relies on **ICU** – International Components for Unicode (see <http://site.icu-project.org/>) – a mature library that fully conforms with the Unicode standard and which provides globalisation support for a broad range of other software applications as well, from web browsers to database systems. Services not covered by **ICU** were implemented from scratch to guarantee that they are as efficient as possible.

Over the years, **stringi** confirmed itself as robust, production-quality software; for many years now it has been one of the most often downloaded R extensions. Interestingly, in 2015 the aforementioned **stringr** package has been rewritten as a set of wrappers around some of the **stringi** functions instead of the base R ones. In Section 14.7 of *R for Data Science* (Wickham and Grolemund 2017) we read: *stringr is useful when you’re learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions. stringi, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need: stringi has 250 functions to stringr’s 49.*

This paper describes the most noteworthy facilities provided by **stringi** that statisticians and data analysts may find useful in their daily activities. We demonstrate how important it is for a modern data scientist to be aware of the challenges of natural language processing in the internet era: how to force “groß” compare equal to “GROSS”, make “a13” ordered before “a100”, or convert between “GRINNING FACE” and “☺” forth and back. As such operations are performed by the very **ICU** itself, we believe that what follows may be of interest to data-oriented practitioners employing Python, PHP, etc., as the library has actively maintained bindings for many other languages.

Here is the outline of the paper.

- Section 2 illustrates the importance of string processing in an example data preparation activity.
- Basic string operations, such as computing length and width of strings, string concatenation, extracting and replacing substrings, are discussed in Section 3.
- Section 4 discusses searching for fixed substrings: counting the number of matches, locating their positions, replacing them with other data, and splitting strings into tokens. We also exemplify how deep vectorisation and the recycling rule with respect to all parameters can be useful in different contexts.

- Section 5 details **ICU** regular expressions, which are a powerful tool for matching patterns defined in a more abstract way, e.g., extracting numbers from text so that they can be processed quantitatively, identifying hyperlinks, etc. We show where **ICU** is different from other libraries like **PCRE**; in particular that it allows for portable, Unicode-correct look-ups, for instance, involving sequences of emojis or mathematical symbols.
- Section 6 deals with the locale-aware **ICU** Collator, which is suitable for natural language processing activities; this is where we exemplify that text processing in different languages or regions may be governed by rules that can turn out surprising to users from other cultural settings. The operations discussed include testing string equivalence (which can turn out useful when we scrape data that consist of non-normalised strings, ignorable punctuation, or accented characters) as well as arranging strings with regards to different linear orderings.
- Section 7 covers some other useful operations such as text boundary analysis (for splitting text into words or sentences), trimming, padding, and other formatting, random string generation, character transliteration (converting between cases and alphabets, removing diacritic marks, etc.) as well as date-time formatting and parsing in any locale (e.g., Japanese dates in a German R).
- Section 8 details on encoding conversion and detection (which is key when reading/writing text files that are to be transformed from/to different systems) as well as Unicode normalisation (which can be useful for removing formatting distinctions from text, e.g., superscripts or font variants).
- Finally, Section 9 closes the paper with a discussion on portability issues with base R, the (high) performance of **stringi**, its C++ API, and ideas for further development.

This paper is by no means a substitute for the comprehensive yet much more technical and in-depth reference manual available by calling `help(package="stringi")`, see also <https://stringi.gagolewski.com/>. Rather, here we explain the package's key design principles and broadly introduce the ideas and services that help program, correct, and optimise text processing workflows.

Let us emphasise that all the below-presented illustrations, i.e., calls to **stringi** functions on different exemplary arguments together with the generated outputs, form an integral part of this manuscript's text. They have been included based on the author's experience-based belief that each "picture" (that we print out below using a monospaced font) is worth hundreds of words.

All the code chunks' outputs presented in this paper were obtained in R 4.0.4. The R environment itself and all the packages used herein are available from CRAN at <https://CRAN.R-project.org/>. In particular, `install.packages("stringi")` can be called to fetch the object of our focus: here we describe **stringi** 1.5.3, which has been built against the following version of the **ICU** library:

```
R> library("stringi") # load and attach the package's namespace
R> cat(stri_info(short=TRUE))
```

```
stringi_1.5.3 (en_AU.UTF-8; ICU4C 61.1 [bundle]; Unicode 10.0)
```

2. Motivational example: Data preparation

Before going into details on the broad array of facilities offered by the **stringi** package itself, let us first demonstrate that string processing is indeed a relevant part of statistical analysis workflows. What follows is a short case study where we prepare a web-scraped data set for further processing.

Assume we wish to gather and analyse climate data for major cities around the world based on information downloaded from *Wikipedia*. For each location from a given list of settlements (e.g., fetched from one of the pages linked under https://en.wikipedia.org/wiki/Lists_of_cities), we would like to harvest the relevant temperature and precipitation data. Without harm in generality, let us focus on the city of Melbourne, VIC, Australia.

The parsing of the city's *Wikipedia* page can be done by means of the functions from the **xml2** (Wickham, Hester, and Ooms 2020) and **rvest** (Wickham 2020) packages.

```
R> library("xml2")
R> library("rvest")
```

First, we load and parse the HTML file downloaded on 2020-09-17 (see the accompanying supplementary files):

```
R> f <- read_html("20200917_wikipedia_melbourne.html")
```

Second, we extract all `table` elements and gather them in a list of HTML nodes, `all_tables`. We then extract the underlying raw text data and store them in a character vector named `text_tables`.

```
R> all_tables <- html_nodes(f, "table")
R> text_tables <- sapply(all_tables, html_text)
R> str(text_tables, nchar.max=65, vec.len=5, strict.width="wrap") # preview
```

```
chr [1:45] "MelbourneVictoriaFrom top, left to right: Flinde"| __truncated__
      "Mean max temp\n Mean min temp\n Annual rainfal"| __truncated__ "This
      section needs additional citations for veri"| __truncated__ "Climate data
      for Melbourne Regional Office (1991"| __truncated__ "Country of Birth
      (2016)[178]Birthplace[N 1]\nPop"| __truncated__ ...
```

Most *Wikipedia* pages related to particular cities include a table labelled as “Climate data”. We need to pinpoint it amongst all the other tables. For this, we will rely on **stringi**'s `stri_detect_fixed()` function that, in the configuration below, is used to extract the index of the relevant table.

```
R> library("stringi")
R> (idx <- which(stri_detect_fixed(text_tables, "climate data",
+   case_insensitive=TRUE, max_count=1)))
```

```
[1] 4
```

Of course, the detailed description of all the facilities brought by **stringi** is covered below. In the meantime, let us use **rvest**'s `html_table()` to convert the above table to a data frame object.

```
R> (x <- html_table(all_tables[[idx]], fill=TRUE))

Climate data for Melbourne Regional Office (1991-2015)
1                                     Month
2                      Record high °C (°F)
3                      Average high °C (°F)
4                      Daily mean °C (°F)
5                      Average low °C (°F)
6                      Record low °C (°F)
7                      Average rainfall mm (inches)
8                      Average rainy days ( 1mm)
9                      Average afternoon relative humidity (%)
10                     Mean monthly sunshine hours
11                     Source: Bureau of Meteorology.[85][86][87]
Climate data for Melbourne Regional Office (1991-2015).1 ...
1                                     Jan ...
2                      45.6(114.1) ...
3                      27.0(80.6) ...
4                      21.6(70.9) ...
5                      16.1(61.0) ...
6                      5.5(41.9) ...
7                      44.2(1.74) ...
8                      5.6 ...
9                      47 ...
10                     279 ...
11                     Source: Bureau of Meteorology.[85][86][87] ...
Climate data for Melbourne Regional Office (1991-2015).3
1                                     Year
2                      46.4(115.5)
3                      20.8(69.4)
4                      16.2(61.2)
5                      11.6(52.9)
6                      -2.8(27.0)
7                      600.9(23.66)
8                      90.6
9                      51
10                     2,191
11                     Source: Bureau of Meteorology.[85][86][87]
```

It is evident that this object requires some significant cleansing and transforming before it can be subject to any statistical analyses. First, for the sake of convenience, let us convert it to a character matrix so that the processing of all the cells can be vectorised (a matrix in R is just a single “long” vector, whereas a data frame is a list of many atomic vectors).

```
R> x <- as.matrix(x)
```

The `as.numeric()` function will find the parsing of the Unicode MINUS SIGN (U+2212, “−”) difficult, therefore let us call the transliterator first in order to replace it (and other potentially problematic characters) with its simpler equivalent:

```
R> x[, ] <- stri_trans_general(x, "Publishing-Any; Any-ASCII")
```

Note that it is the first row of the matrix that defines the column names. Moreover, the last row just gives the data source and hence may be removed.

```
R> dimnames(x) <- list(x[, 1], x[1, ]) # row, column names
R> x <- x[2:(nrow(x)-1), 2:ncol(x)]    # skip 1st/last row and 1st column
R> x[, c(1, ncol(x))] # example columns
```

	Jan	Year
Record high °C (°F)	"45.6(114.1)"	"46.4(115.5)"
Average high °C (°F)	"27.0(80.6)"	"20.8(69.4)"
Daily mean °C (°F)	"21.6(70.9)"	"16.2(61.2)"
Average low °C (°F)	"16.1(61.0)"	"11.6(52.9)"
Record low °C (°F)	"5.5(41.9)"	"-2.8(27.0)"
Average rainfall mm (inches)	"44.2(1.74)"	"600.9(23.66)"
Average rainy days (>= 1mm)	"5.6"	"90.6"
Average afternoon relative humidity (%)	"47"	"51"
Mean monthly sunshine hours	"279"	"2,191"

Commas that are used as thousands separators (commas surrounded by digits) should be dropped:

```
R> x[, ] <- stri_replace_all_regex(x, "(?<=\\d),(?\\d)", "")
```

The numbers and alternative units in parentheses are redundant, therefore these should be taken care of as well:

```
R> x[, ] <- stri_replace_all_regex(x,
+   "\\d+(?:\\.\\d+)?\\((\\d+(?:\\.\\d+)?\\)\\)", "$1")
R> dimnames(x)[[1]] <- stri_replace_all_fixed(dimnames(x)[[1]],
+   c(" (°F)", " (inches)"), c("", ""))
R> vectorise_all=FALSE
```

At last, `as.numeric()` can be used to re-interpret all the strings as numbers:

```
R> x <- structure(as.numeric(x), dim=dim(x), dimnames=dimnames(x))
R> x[, c(1, 6, ncol(x))] # example columns
```

	Jan	Jun	Year
Record high °C	45.6	22.4	46.4
Average high °C	27.0	15.1	20.8
Daily mean °C	21.6	11.7	16.2
Average low °C	16.1	8.2	11.6
Record low °C	5.5	-2.2	-2.8
Average rainfall mm	44.2	49.5	600.9
Average rainy days (≥ 1 mm)	5.6	8.6	90.6
Average afternoon relative humidity (%)	47.0	61.0	51.0
Mean monthly sunshine hours	279.0	108.0	2191.0

We now have a cleansed matrix at our disposal. We can, for instance, compute the monthly temperature ranges:

```
R> x["Record high °C", -ncol(x)] - x["Record low °C", -ncol(x)]
```

```
  Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
40.1 41.9 38.9 33.4 29.8 24.6 26.1 28.6 31.9 36.8 38.4 39.3
```

or the average daily precipitation:

```
R> sum(x["Average rainfall mm", -ncol(x)]) / 365.25
```

```
[1] 1.6463
```

and so forth.

For the climate data on other cities, very similar need to be performed – the whole process of scraping and cleansing data can quite easily be automated, perhaps with some minor adjustments. The above functions are not only convenient to use, but also efficient and portable across different platforms.

3. Basic string operations

Let us proceed with a detailed description of the most important facilities in the **stringi** package that might be of interest to the broad statistical and data analysis audience.

3.1. Benefits of vectorisation

First and foremost, we should emphasise that the R language itself does not feature any classical scalar types. Individual character strings are wrapped around atomic vectors of type **character**:

```
R> "spam"           # or 'spam'
```

```
[1] "spam"
```

```
R> typeof("spam")
```

```
[1] "character"
```

```
R> length("spam") # a character vector of length 1 - a single string
```

```
[1] 1
```

Not having a separate scalar type is very convenient; the so-called *vectorisation* strategy encourages the writing of code that processes *whole* collections of objects, all at once, regardless of their size. For example, let us consider the following data frame:

```
R> (birth_dates <-  
+   head(read.csv("birth_dates.csv", header=TRUE, comment.char="#"), n=3))
```

```
      Name BirthDate  
1 Eckehard Grünfeld 01.11.1911  
2   Zbyněk Slavík 29.03.1966  
3   Marx Crowell 17.03.2009
```

The `Name` column is of type `character`. For instance (with all the details provided in a section to follow), here is how we can separate the first and the last names from each other (assuming for simplicity that no middle names are given), using just a single function call:

```
R> (birth_names <- stri_split_fixed(birth_dates$Name, " ",  
+   n=2, simplify=TRUE))
```

```
      [,1]      [,2]  
[1,] "Eckehard" "Grünfeld"  
[2,] "Zbyněk"   "Slavík"  
[3,] "Marx"     "Crowell"
```

Due to vectorisation, we can generally avoid using `for/while` loops, which makes the code much more readable, maintainable, and faster to execute.

3.2. Computing length and width

First we shall review the functions related to counting the number of *entities* in each string.

Length. Let us consider the following character vector:

```
R> x <- c("spam", "bacon", "", "sausage", NA, "spam")  
R> length(x) # vector length
```

```
[1] 6
```


`stri_length()` computes the *length* of each string. More precisely, the function gives the number of Unicode code points in each string, see Section 8.1 for more details.

```
R> stri_length(x)

[1] 4 5 0 7 NA 4
```

`stri_length(x)` returns a numeric vector `s`, with the same number of elements as `x`, such that, for every `i`, `s[i]` is the length of the string `x[i]`. Note that the 3rd element in `x` is an empty string, `"`, hence its length is 0. Moreover, there is a missing (NA) value at index 5, therefore the corresponding length is *undefined* as well. **stringi** applies this rule consistently across all its functions.

Zero-length. To quickly determine which of the items are empty strings, we may call:

```
R> stri_isempty(x)

[1] FALSE FALSE TRUE FALSE NA FALSE
```

Note that we distinguish between an empty character vector (`character(0)`, i.e., a zero-length sequence) and an empty string, i.e., a string of length 0 wrapped in a vector of length 1.

```
R> length(character(0))      # no strings at all

[1] 0
```

```
R> stri_length(character(0))

integer(0)
```

```
R> length("")               # one empty string

[1] 1
```

```
R> stri_length("")

[1] 0
```

Width. Sometimes merely knowing the number of characters in a string is not enough. For instance, when preparing a formatted output (e.g., in a report dynamically generated with **Sweave()** or **knitr**; see Xie 2015), a string's *width* estimate – an approximate number of text columns it occupies when printed using a monospaced font – may be more informative. In particular, many Chinese, Japanese, Korean, or emoji characters take up two text cells. Some code points, on the other hand, are of width 0 (e.g., the ZERO WIDTH SPACE, U+200B).

```
R> stri_length(c(" 你好", "\u200b\u200b\u200b\u200b", "123456"))
```

```
[1] 2 4 6
```

The first string (a greeting) consists of 2 Chinese characters (U+4F60, U+597D), the second is comprised of 4 zero-width spaces, and the third one carries 6 ASCII digits. Here are their corresponding widths:

```
R> stri_width(c(" 你好", "\u200b\u200b\u200b\u200b", "123456"))
```

```
[1] 4 0 6
```

3.3. Joining

Below we describe the functions that are related to string concatenation.

Operator %s+%. To join the corresponding strings in two character vectors, we may use the binary %s+% operator:

```
R> "tasty" %s+% "spam"
```

```
[1] "tastyspam"
```

```
R> c("tasty", "delicious") %s+% c("spam", "bacon") # elementwise
```

```
[1] "tastyspam"      "deliciousbacon"
```

```
R> c("tasty", "delicious") %s+% "spam" # recycling rule
```

```
[1] "tastyspam"      "deliciousspam"
```

```
R> c("tasty", "delicious", "savoury", "yummy") %s+% c("spam", "bacon")
```

```
[1] "tastyspam"      "deliciousbacon" "savouryspam"    "yummybacon"
```

```
R> character(0) %s+% c("spam", "bacon")
```

```
character(0)
```

This operator is vectorised in exactly the same manner as other arithmetic operators in R. In particular, the recycling rule is used if the inputs are of different lengths. If one of the arguments is empty, the result is a zero-length vector as well. Moreover, **stringi** does enforce the consistent propagation of missing values (unlike the built-in `paste()` function):

```
R> x <- c("spam", "bacon", "", "sausage", NA, "spam")
```

```
R> x %s+% "!"
```

```
[1] "spam!"      "bacon!"      "!"           "sausage!" NA           "spam!"
```

For dealing with missing values, we may rely on the convenience functions such as `stri_replace_na()`, `stri_omit_na()`, and if additionally we would like to get rid of empty strings in a vector, `stri_omit_empty_na()`:

```
R> stri_replace_na(x, "<NA>") %s+% "!"
```

```
[1] "spam!"      "bacon!"      "!"           "sausage!" "<NA>!"      "spam!"
```

```
R> stri_omit_empty_na(x) %s+% "!"
```

```
[1] "spam!"      "bacon!"      "sausage!" "spam!"
```

Flattening. The elements in a character vector can be joined altogether to form a single string via a call to `stri_flatten()`:

```
R> stri_flatten(stri_omit_empty_na(x)) # collapse="" by default
```

```
[1] "spambaconsausagespam"
```

```
R> stri_flatten(stri_omit_empty_na(x), collapse=", ")
```

```
[1] "spam, bacon, sausage, spam"
```

Generalisation. Both the `%s+%` operator and the `stri_flatten()` function are generalised by `stri_join()` (alias: `stri_paste()`, `stri_c()`):

```
R> stri_join(c("X", "Y", "Z"), 1:6, "a") # sep="", collapse=NULL
```

```
[1] "X1a" "Y2a" "Z3a" "X4a" "Y5a" "Z6a"
```

```
R> stri_join(c("X", "Y", "Z"), 1:6, "a", sep="_", collapse=", ")
```

```
[1] "X_1_a, Y_2_a, Z_3_a, X_4_a, Y_5_a, Z_6_a"
```

Note how the two (1st, 3rd) shorter vectors were recycled to match the longest vector's (2nd) length. The latter was of numeric type, but it was implicitly coerced with a call to `as.character()`. More examples:

```
R> stri_join(birth_names[,2], birth_names[,1], sep=", ")
```

```
[1] "Grünfeld, Eckehard" "Slavík, Zbyněk"      "Crowell, Marx"
```

```
R> outer(LETTERS[1:3], 1:5, stri_join, sep=".") # outer product
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,] "A.1" "A.2" "A.3" "A.4" "A.5"
[2,] "B.1" "B.2" "B.3" "B.4" "B.5"
[3,] "C.1" "C.2" "C.3" "C.4" "C.5"
```

Duplicating. To duplicate given strings, we call `stri_dup()` or the `%s*%` operator:

```
R> stri_dup(letters[1:5], 2)
```

```
[1] "aa" "bb" "cc" "dd" "ee"
```

```
R> stri_dup("spam", 1:3)
```

```
[1] "spam"          "spamsam"       "spamsamsam"
```

```
R> stri_dup(letters[1:3], 1:3)
```

```
[1] "a"   "bb"  "ccc"
```

```
R> "a" %s*% 5
```

```
[1] "aaaaa"
```

Again, we see a vectorisation with regards to all the arguments.

Within-list joining. There is also a convenience function that applies `stri_flatten()` on each character vector in a given list:

```
R> words <- list(c("spam", "bacon", "sausage", "spam"), c("eggs", "spam"))
```

```
R> stri_join_list(words, sep=",")
```

```
[1] "spam,bacon,sausage,spam" "eggs,spam"
```

```
R> stri_join_list(words, sep=";", collapse="; ")
```

```
[1] "spam,bacon,sausage,spam; eggs,spam"
```

We shall see that such lists of strings are generated by `stri_sub_all()`, `stri_extract_all()`, and similar functions.

3.4. Extracting and replacing substrings

Next group of functions deals with the extraction and replacement of particular sequences of code points in given strings.

Indexing vectors. Recall that in order to select a subsequence from any R vector, we use the square-bracket operator¹ with an index vector consisting of either non-negative integers, negative integers, or logical values².

```
R> x[1:3]                                # from 1st to 3rd string

[1] "spam" "bacon" ""

R> x[c(1, length(x))]                   # 1st and last

[1] "spam" "spam"

R> x[-1]                                # all but 1st

[1] "bacon" "" "sausage" NA "spam"

R> x[!stri_isempty(x) & !is.na(x)]      # filtering based on a logical vector

[1] "spam" "bacon" "sausage" "spam"
```

Extracting substrings. A character vector is, in its very own essence, a sequence of sequences of code points. To extract specific substrings from each string in a collection, we can use the `stri_sub()` function.

```
R> y <- "spam, egg, spam, spam, bacon, and spam"
R> stri_sub(y, 18)                       # from 18th code point to end

[1] "spam, bacon, and spam"

R> stri_sub(y, 12, to=15)                 # from 12th to 15th code point (inclusive)

[1] "spam"

R> stri_sub(y, 12, length=4)             # 4 code points from 12th

[1] "spam"
```

Moreover, negative indices count from the end of a string.

```
R> stri_sub(y, -15)                      # from 15th last to end

[1] "bacon, and spam"
```

¹More precisely, `x[i]` is a syntactic sugar for a call to ``[` (x, i)`. Moreover, if `x` is a list, `x[[i]]` can be used to extract its *i*-th element (alias ``[[` (x, i)`). Knowing the “functional” form of the operators allows us to, for instance, extract all first elements from each vector in a list by simply calling `apply(x, "[", 1)`.

²If an object’s `names` attribute is set, indexing with a character vector is also possible.

stri_sub_all() function. The `stri_sub()` function is of course vectorised with respect to all its arguments (the subsetting character vector, `from`, and `to` or `length`). If one of the vectors is of smaller length than the other ones, the recycling rule is applied as usual. For instance:

```
R> stri_sub(y, c(1, 12, 18), length=4) # different substrings of one string

[1] "spam" "spam" "spam"

R> stri_sub(x[c(1, 2, 4)], from=-3)    # same substrings of different strings

[1] "pam" "con" "age"

R> stri_sub(x[c(1, 2, 4)],
+   c(-4, -2, -5)) # different substrings of different strings

[1] "spam" "on" "usage"
```

If some deeper vectorisation level is necessary, `stri_sub_all()` comes in handy. It allows to extract multiple (possibly different) substrings from all the strings provided:

```
R> (z <- stri_sub_all(x[c(1, 2, 4)],
+   from= list(c(1, 3, 4), -2, c(1, 4)),
+   length=list(1, 2, c(4, 3))))

[[1]]
[1] "s" "a" "m"

[[2]]
[1] "on"

[[3]]
[1] "saus" "sag"
```

As the number of substrings to extract from each string might vary, the result is a list of character strings. These may all be concatenated by means of the above-mentioned `stri_join_list()` function.

```
R> stri_join_list(z, sep=", ")

[1] "s, a, m" "on" "saus, sag"
```

On a side note, there is also a more flexible version of the built-in `simplify2array()` function whose aim is to convert such lists to matrices.

```
R> stri_list2matrix(z)
```

```

      [,1] [,2] [,3]
[1,] "s"  "on" "saus"
[2,] "a"  NA  "sag"
[3,] "m"  NA  NA

```

```
R> stri_list2matrix(z, byrow=TRUE, fill="", n_min=5)
```

```

      [,1] [,2] [,3] [,4] [,5]
[1,] "s"   "a"   "m"   ""   ""
[2,] "on"  ""    ""    ""   ""
[3,] "saus" "sag" ""    ""   ""

```

Again, let us note that no explicit `for/while` loops are necessary. For example, here is a way to extract non-consecutive substrings from each string – ones that consist of the first and the last letter:

```
R> stri_join_list(stri_sub_all(x[c(1, 2, 4)], c(1, -1), length=1))
```

```
[1] "sm" "bn" "se"
```

Permuting code points. Somehow related to the above are different ways to construct various permutations (possibly with replacement) of code points in a string:

```
R> stri_join_list(stri_sub_all("spam", c(4, 3, 2, 3, 1), length=1))
```

```
[1] "mapas"
```

```
R> stri_rand_shuffle("bacon") # random order
```

```
[1] "anobc"
```

```
R> stri_reverse("spam") # reverse order
```

```
[1] "maps"
```

from_to matrices. The second parameter of both `stri_sub()` and `stri_sub_list()` can also be fed with a two-column matrix of the form `cbind(from, to)`. Here, the first column gives the start indices and the second column defines the end ones. Such matrices are generated, amongst others, by the `stri_locate_*`() functions (see below for details).

```
R> (from_to <- cbind(from=c(1, 12, 18), to=c(4, 15, 21))) # +optional labels
```

```

      from to
[1,]    1  4
[2,]   12 15
[3,]   18 21

```

```
R> stri_sub(y, from_to)
```

```
[1] "spam" "spam" "spam"
```

Another example (the recycling rule):

```
R> (from_to <- matrix(1:8, ncol=2, byrow=TRUE))
```

```
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
```

```
R> stri_sub(c("abcdefgh", "ijklmnop"), from_to)
```

```
[1] "ab" "kl" "ef" "op"
```

Now let us note the difference between the above output and the following one:

```
R> stri_sub_all(c("abcdefgh", "ijklmnop"), from_to)
```

```
[[1]]
[1] "ab" "cd" "ef" "gh"

[[2]]
[1] "ij" "kl" "mn" "op"
```

Replacing substrings. `stri_sub_replace()` returns a version of a character vector with each specified chunk replaced by another string:

```
R> stri_sub_replace(c("abcde", "ABCDE"),
+   from=c(2, 4), length=c(1, 2), replacement=c("X", "Y"))
```

```
[1] "aXcde" "ABCY"
```

```
R> stri_sub_replace("abcde",
+   from=c(2, 4), length=1, replacement=c("X", "Y"))
```

```
[1] "aXcde" "abcYe"
```

Similarly, `stri_sub_replace_all()` allows for replacing multiple substrings within each component of a character vector:

```
R> stri_sub_replace_all(c("abcde", "ABCDE"),
+   from=c(2, 4), length=1, replacement=c("X", "Y"))
```



```
[1] "aXcYe" "AXCYE"
```

```
R> stri_sub_replace_all("abcde",
+   from=c(2, 4), length=1, replacement=c("X", "Y"))
```

```
[1] "aXcYe"
```

```
R> stri_sub_replace_all(c("abcde", "ABCDE"),
+   from=list(c(2, 4), c(1, 3)), length=list(1, c(1, 2)),
+   replacement=list("Z", c("XX", "YYYYY")))
```

```
[1] "aZcZe"      "XXBYYYYYE"
```

Replacing substrings in-place. The corresponding replacement functions allow for modifying a character vector in-place:

```
R> y2 <- y
R> stri_sub(y2, 7, length=3) <- "spam" # in-place replacement, egg → spam
R> print(y2)                           # y2 has been changed
```

```
[1] "spam, spam, spam, spam, bacon, and spam"
```

```
R> y3 <- "a b c"
R> stri_sub_all(y3, c(1, 3, 5), length=1) <- c("A", "B", "C")
R> print(y3)                           # y3 has been changed
```

```
[1] "A B C"
```

4. Code-pointwise comparing of strings

There are many situations where we are faced with testing whether two strings (or parts thereof) consist exactly of the same Unicode³ code points, in the same order. These include, for instance, matching a nucleotide sequence in a DNA profile and querying for system resources based on file names or UUIDs. Such tasks, due to their simplicity, can be performed very efficiently.

4.1. Testing for equality of strings

To quickly test whether the corresponding strings in two character vectors are identical (in a code-pointwise manner), we can use the `%s==%` operator or the `stri_cmp_eq()` function.

```
R> "actg" %s==% c("ACTG", "actg", "act", NA) # recycling rule
[1] FALSE TRUE FALSE NA
```

Moreover, `%s!=%` and `stri_cmp_neq()`, respectively, are their negations.

4.2. Searching for fixed strings

Table 1 lists the string search functions available in **stringi**. Below we explain their behaviour in the context of fixed pattern matching. Notably, their description is very detailed, because – as we shall soon find out – similar search functions are available for the other string matching engines (namely, those relying on regular expressions and the **ICU** Collator, see Section 5 and Section 6).

For detecting if a string contains a given substring (code-pointwisely), the fast KMP (Knuth, Morris, and Pratt 1977) search algorithm, with worst time complexity of $O(n + p)$ (where n is the length of the string and p is the length of the pattern), has been implemented in **stringi** (with numerous tweaks for even faster matching).

4.3. Counting matches

The `stri_count_fixed()` function counts the number of times a fixed pattern occurs in a given string.

```
R> stri_count_fixed("abcabcdefabcabcbdc", "abc") # search pattern is "abc"
[1] 4
```

Equivalently, we can call the more generic (see below) function `stri_count()` with the `fixed=pattern` argument:

```
R> stri_count("abcabcdefabcabcbdc", fixed="abc")
[1] 4
```

³All functions in **stringi** automatically convert all R strings to Unicode, see Section 8.2 for a discussion.

Name(s)	Meaning
<code>stri_count()</code>	count pattern matches
<code>stri_detect()</code>	detect pattern matches
<code>stri_endswith()</code>	[all but <code>regex</code>] detect pattern matches at end of string
<code>stri_extract_all()</code> , <code>stri_extract_first()</code> , <code>stri_extract_last()</code>	extract pattern matches
<code>stri_locate_all()</code> , <code>stri_locate_first()</code> , <code>stri_locate_last()</code>	locate pattern matches
<code>stri_match_all()</code> , <code>stri_match_first()</code> , <code>stri_match_last()</code>	[<code>regex</code> only] extract matches to regex capture groups
<code>stri_replace_all()</code> , <code>stri_replace_first()</code> , <code>stri_replace_last()</code>	substitute pattern matches with a replacement string
<code>stri_split()</code>	split up a string at pattern matches
<code>stri_startswith()</code>	[all but <code>regex</code>] detect pattern matches at start of string
<code>stri_subset()</code> , <code>`stri_subset<-`()</code>	return or replace strings that contain pattern matches

Table 1: String search/pattern matching functions in **stringi**. Each function, unless otherwise indicated, can be used in conjunction with any search engine, e.g., we have `stri_count_fixed()` (see Section 4), `stri_detect_regex()` (see Section 5), and `stri_split_coll()` (see Section 6).

4.4. Benefits of recycling rule

All the string search functions are vectorised with respect to both the *haystack* and the *needle* arguments (and, e.g., the *replacement* string, if applicable). As usual, the shorter vector is recycled if necessary. Some users, unaware of this rule, might find this behaviour unintuitive at the beginning. Therefore, let us enumerate the most useful scenarios that are possible thanks to the arguments' recycling:

- many strings – one pattern:

```
R> stri_count_fixed(c("abc", "abcd", "abcabc", "abdc", "dab", NA), "abc")
[1] 1 1 2 0 0 NA
```

- one string – many patterns:

```
R> stri_count_fixed("abc", c("def", "bc", "abc", "abcde", NA))
```

```
[1] 0 1 1 0 NA
```

- each string – its own corresponding pattern:

```
R> stri_count_fixed(c("abc", "def", "ghi"), c("a", "z", "h"))
```

```
[1] 1 0 1
```

- each row in a matrix – its own corresponding pattern:

```
R> (A <- matrix(
+   do.call(stri_paste,
+     expand.grid(
+       c("a", "b", "c"), c("a", "b", "c"), c("a", "b", "c")
+     )),
+   nrow=3))
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,] "aaa" "aba" "aca" "aab" "abb" "acb" "aac" "abc" "acc"
[2,] "baa" "bba" "bca" "bab" "bbb" "bcb" "bac" "bbc" "bcc"
[3,] "caa" "cba" "cca" "cab" "cbb" "ccb" "cac" "cbc" "ccc"
```

```
R> matrix(stri_count_fixed(A, c("a", "b", "c")), nrow=3)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[1,]     3     2     2     2     1     1     2     1     1
[2,]     1     2     1     2     3     2     1     2     1
[3,]     1     1     2     1     1     2     2     2     3
```

The above is due to the fact that matrices are represented as “flat” vectors of length `length nrow(A)*ncol(A)`, whose elements are read in a column-major (Fortran) order. Therefore, in the above example, pattern “a” is being sought in the 1st, 4th, 7th, ... string in A, i.e., “aaa”, “aba”, “aca”, ...; pattern “b” in the 2nd, 5th, 8th, ... string; and “c” in the 3rd, 6th, 9th, ... one.

On a side note, to match different patterns with respect to each *column*, we can (amongst others) apply matrix transpose twice (`t(stri_count_fixed(haystack, t(needle)))`) or use the `rep()` function to properly replicate the *needles*:

```
R> (At <- t(A)) # example haystack
```

```
      [,1] [,2] [,3]
[1,] "aaa" "baa" "caa"
[2,] "aba" "bba" "cba"
[3,] "aca" "bca" "cca"
[4,] "aab" "bab" "cab"
[5,] "abb" "bbb" "cbb"
[6,] "acb" "bcb" "ccb"
```

```
[7,] "aac" "bac" "cac"
[8,] "abc" "bbc" "cbc"
[9,] "acc" "bcc" "ccc"
```

```
R> matrix(stri_count_fixed(At, rep(c("a", "b", "c"), each=nrow(At))), ncol=3)
```

```
      [,1] [,2] [,3]
[1,]     3     1     1
[2,]     2     2     1
[3,]     2     1     2
[4,]     2     2     1
[5,]     1     3     1
[6,]     1     2     2
[7,]     2     1     2
[8,]     1     2     2
[9,]     1     1     3
```

A similar search in the case of a data frame-type input (any list of character vectors of identical lengths) can be performed by means of a call to `mapply()`:

```
R> (At.df <- as.data.frame(At))
```

```
      V1 V2 V3
1 aaa baa caa
2 aba bba cba
3 aca bca cca
4 aab bab cab
5 abb bbb cbb
6 acb bcb ccb
7 aac bac cac
8 abc bbc cbc
9 acc bcc ccc
```

```
R> mapply(stri_count_fixed, At.df, c("a", "b", "c"))
```

```
      V1 V2 V3
[1,]  3  1  1
[2,]  2  2  1
[3,]  2  1  2
[4,]  2  2  1
[5,]  1  3  1
[6,]  1  2  2
[7,]  2  1  2
[8,]  1  2  2
[9,]  1  1  3
```

- all strings – all patterns:

```
R> x <- c("aaa", "bbb", "ccc", "abc", "cba", "aab", "bab", "acc")
R> y <- c("a", "b", "c")
R> structure(
+   outer(x, y, stri_count_fixed),
+   dimnames=list(x, y) # add row and column names
+ )
```

```
      a b c
aaa 3 0 0
bbb 0 3 0
ccc 0 0 3
abc 1 1 1
cba 1 1 1
aab 2 1 0
bab 1 2 0
acc 1 0 2
```

A similar result (without the post-processing of the return value, which can be done through a call to `matrix()`) may be obtained by calling:

```
R> stri_count_fixed(rep(x, each=length(y)), y)

[1] 3 0 0 0 3 0 0 0 3 1 1 1 1 1 1 2 1 0 1 2 0 1 0 2
```

4.5. Search engine options

The pattern matching engine may be tuned up by passing further arguments to the search functions (via "..."; they are be redirected as-is to `stri_opts_fixed()`). Table 2 gives the list of available options.

First, we may switch on the simplistic⁴ case-insensitive matching.

```
R> stri_count_fixed("ACTGACGacgggACg", "acg", case_insensitive=TRUE)

[1] 3
```

Second, we can indicate whether we are interested in detecting overlapping pattern matches or whether searching should continue at the end of each match (the latter being the default behaviour):

```
R> stri_count_fixed("acatgacaca", "aca") # overlap=FALSE (default)

[1] 2

R> stri_count_fixed("acatgacaca", "aca", overlap=TRUE)

[1] 3
```

Option	Purpose
<code>case_insensitive</code>	logical; whether to enable the simple case-insensitive matching (defaults to <code>FALSE</code>)
<code>overlap</code>	logical; whether to enable the detection of overlapping matches (defaults to <code>FALSE</code>); available in <code>stri_extract_all_fixed()</code> , <code>stri_locate_all_fixed()</code> , and <code>stri_count_fixed()</code>

Table 2: Options for the fixed pattern search engine, see `stri_opts_fixed()`.

4.6. Detecting and subsetting patterns

A somehow simplified version of the above search task involves asking whether a pattern occurs in a string at all. Such an operation can be performed with a call to `stri_detect_fixed()`.

```
R> x <- c("abc", "abcd", "def", "xyzabc", "uabdc", "dab", NA, "abc")
R> stri_detect_fixed(x, "abc")
```

```
[1] TRUE TRUE FALSE TRUE FALSE FALSE NA TRUE
```

We can also indicate that a no-match is rather of our interest:

```
R> stri_detect_fixed(x, "abc", negate=TRUE)
```

```
[1] FALSE FALSE TRUE FALSE TRUE TRUE NA FALSE
```

What is more, there is an option to stop searching once a given number of matches has been found in the `haystack` vector (as a whole), which can speed up the processing of larger data sets:

```
R> stri_detect_fixed(x, "abc", max_count=3)
```

```
[1] TRUE TRUE FALSE TRUE NA NA NA NA
```

```
R> stri_detect_fixed(x, "abc", negate=TRUE, max_count=2)
```

```
[1] FALSE FALSE TRUE FALSE TRUE NA NA NA
```

This can be useful in scenarios such as “find the first 5 matching resource IDs”.

There are also functions that verify whether a string starts or ends⁵ with a pattern match:

⁴Which is not suitable for real-world NLP tasks, as it assumes that changing the case of a single code point always produces one and only one item; This way, “groß” does not compare equal to “GROSS”, see Section 6 (and partially Section 5) for a workaround.

⁵Note that testing for a pattern match at the start or end of a string has not been implemented separately for `regex` patterns, which support “^” and “\$” anchors that serve exactly this very purpose.

```
R> stri_startswith_fixed(x, "abc") # from=1 - match at start
```

```
[1] TRUE TRUE FALSE FALSE FALSE FALSE NA TRUE
```

```
R> stri_endswith_fixed(x, "abc") # to=-1 - match at end
```

```
[1] TRUE FALSE FALSE TRUE FALSE FALSE NA TRUE
```

Pattern detection is often performed in conjunction with character vector subsetting. This is why we have a specialised (and hence slightly faster) function that returns only the strings that match a given pattern:

```
R> stri_subset_fixed(x, "abc")
```

```
[1] "abc" "abcd" "xyzabc" NA "abc"
```

The above is equivalent to `x[stri_detect_fixed(x, "abc")]`, but avoids writing `x` twice (convenient when `x` is generated programmatically on the fly, using some complicated expression). Moreover:

```
R> stri_subset_fixed(x, "abc", omit_na=TRUE)
```

```
[1] "abc" "abcd" "xyzabc" "abc"
```

```
R> stri_subset_fixed(x, "abc", negate=TRUE) # everything but the matches
```

```
[1] "def" "uabdc" "dab" NA
```

There is also a replacement version of this function:

```
R> stri_subset_fixed(x, "abc") <- "" # modifies x in-place
```

```
R> x
```

```
[1] "" "" "def" "" "uabdc" "dab" NA ""
```

4.7. Locating and extracting patterns

The functions from the `stri_locate()` family aim to pinpoint the positions of the matches to a pattern. First, we may be interested in the location of the first or the last pattern match:

```
R> x <- c("aga", "actg", NA, "ggAGAGAGaGAca", "agagagaga")
```

```
R> stri_locate_first_fixed(x, "aga")
```



```

      start end
[1,]      1  3
[2,]     NA NA
[3,]     NA NA
[4,]     NA NA
[5,]      1  3

```

```
R> stri_locate_last_fixed(x, "aga")
```

```

      start end
[1,]      1  3
[2,]     NA NA
[3,]     NA NA
[4,]     NA NA
[5,]      7  9

```

In both examples we obtain a two-column (“from–to”) matrix with the number of rows determined by the recycling rule (here: the length of *x*). Missing values correspond to either missing inputs or no-matches.

Second, we may be yearning for the locations of all the matching substrings. As the number of possible answers may differ from string to string, the result is a list of “from–to” matrices.

```
R> stri_locate_all_fixed(x, "aga")
```

```

[[1]]
      start end
[1,]      1  3

[[2]]
      start end
[1,]     NA NA

[[3]]
      start end
[1,]     NA NA

[[4]]
      start end
[1,]     NA NA

[[5]]
      start end
[1,]      1  3
[2,]      5  7

```

Note that, for compatibility with **stringr**, a no-match is indicated by a single-row matrix with two missing values. This behaviour can be changed by setting the `omit_no_match` argument to `TRUE`. Here is an example that additionally asks for overlapping, case insensitive matches:

```
R> stri_locate_all_fixed(x, "aga", omit_no_match=TRUE,
+   overlap=TRUE, case_insensitive=TRUE)
```

```
[[1]]
      start end
[1,]      1   3
```

```
[[2]]
      start end
```

```
[[3]]
      start end
[1,]    NA  NA
```

```
[[4]]
      start end
[1,]      3   5
[2,]      5   7
[3,]      7   9
[4,]      9  11
```

```
[[5]]
      start end
[1,]      1   3
[2,]      3   5
[3,]      5   7
[4,]      7   9
```

Let us recall that “from-to” matrices of this kind constitute particularly convenient inputs to `stri_sub()` and `stri_sub_all()`. However, if merely the extraction of the matching substrings is needed, we can rely on the functions from the `stri_extract()` family:

```
R> (res <- stri_extract_first_fixed(x, "aga", case_insensitive=TRUE))
```

```
[1] "aga" NA    NA    "AGA" "aga"
```

```
R> identical(res, stri_sub(x,
+   stri_locate_first_fixed(x, "aga", case_insensitive=TRUE)))
```

```
[1] TRUE
```

```
R> (res <- stri_extract_all_fixed(x, "aga",
+   overlap=TRUE, case_insensitive=TRUE, omit_no_match=TRUE))
```

```
[[1]]
[1] "aga"
```

```

[[2]]
character(0)

[[3]]
[1] NA

[[4]]
[1] "AGA" "AGA" "Aga" "aGA"

[[5]]
[1] "aga" "aga" "aga" "aga"

R> identical(res, stri_sub_all(x,
+   stri_locate_all_fixed(x, "aga",
+   omit_no_match=TRUE, overlap=TRUE, case_insensitive=TRUE)))

[1] TRUE

```

4.8. Replacing pattern occurrences

In order to replace each matching substring with a corresponding replacement string, we can refer to `stri_replace()`:

```

R> x <- c("aga", "actg", NA, "ggAGAGAgAca", "agagagaga")
R> stri_replace_first_fixed(x, "aga", "~", case_insensitive=TRUE)

[1] "~"          "actg"        NA            "gg~GAgAca"  "~gagaga"

R> stri_replace_last_fixed(x, "aga", "~", case_insensitive=TRUE)

[1] "~"          "actg"        NA            "ggAGAGAg~ca" "agagag~"

R> stri_replace_all_fixed(x, "aga", "~", case_insensitive=TRUE)

[1] "~"          "actg"        NA            "gg~G~GAcA"  "~g~ga"

```

Note that the inputs that are not part of any match are left unchanged.

The function is vectorised with respect to all the three arguments (*haystack*, *needle*, *replacement string*), with the usual recycling behaviour if necessary. If a different arguments' vectorisation scheme is required, we set the `vectorise_all` argument of `stri_replace_all()` to `FALSE`. Compare the following:

```

R> stri_replace_all_fixed("The quick brown fox jumped over the lazy dog.",
+   c("quick", "brown", "fox", "lazy", "dog"),
+   c("slow", "yellow-ish", "hen", "spamity", "lama"))

```

```
[1] "The slow brown fox jumped over the lazy dog."
[2] "The quick yellow-ish fox jumped over the lazy dog."
[3] "The quick brown hen jumped over the lazy dog."
[4] "The quick brown fox jumped over the spamity dog."
[5] "The quick brown fox jumped over the lazy lama."
```

```
R> stri_replace_all_fixed("The quick brown fox jumped over the lazy dog.",
+   c("quick", "brown",      "fox", "lazy", "dog"),
+   c("slow",  "yellow-ish", "hen", "spamity", "lama"),
+   vectorise_all=FALSE)
```

```
[1] "The slow yellow-ish hen jumped over the spamity lama."
```

Here, for every string in the *haystack*, we observe the vectorisation *independently* over the *needles* and replacement strings. Each occurrence of the 1st needle is substituted with the 1st replacement string, then the search is repeated for the 2nd needle in order to replace it with the 2nd corresponding string, and so forth.

4.9. Splitting

To split each element in the *haystack* into substrings, where the *needles* define the delimiters that separate the inputs into tokens, we call `stri_split()`:

```
R> x <- c("a,b,c,d", "e", "", NA, "f,g,,h,i,,j,")
R> stri_split_fixed(x, ",")
```

```
[[1]]
[1] "a" "b" "c" "d"

[[2]]
[1] "e"

[[3]]
[1] ""

[[4]]
[1] NA

[[5]]
[1] "f" "g" "" "" "h" "i" "" "j" ""
```

The result is a list of character vectors, as each string in the *haystack* might be split into a possibly different number of tokens.

There are also options to omit empty strings from the resulting vectors, or limit the number of tokens.

```
R> stri_split_fixed(x, ",", n=3) # stringr compatibility mode
```

```
[[1]]  
[1] "a"  "b"  "c,d"
```

```
[[2]]  
[1] "e"
```

```
[[3]]  
[1] ""
```

```
[[4]]  
[1] NA
```

```
[[5]]  
[1] "f"      "g"      ",,h,i,,j,"
```

```
R> stri_split_fixed(x, ",", n=3, tokens_only=TRUE, omit_empty=TRUE)
```

```
[[1]]  
[1] "a" "b" "c"
```

```
[[2]]  
[1] "e"
```

```
[[3]]  
character(0)
```

```
[[4]]  
[1] NA
```

```
[[5]]  
[1] "f" "g" "h"
```

5. Regular expressions

Regular expressions (*regexes*) provide us with a concise grammar for defining systematic patterns which can be sought in character strings, in particular:

1. specific substrings,
2. emojis of any kind,
3. stand-alone sequences of lower-case Latin letters (“words”),
4. substrings that can be interpreted as real numbers (with or without fractional part, also in scientific notation),
5. telephone numbers,
6. email addresses, or
7. URLs.

Theoretically, the concept of matching regular patterns dates back to the so-called regular languages and finite state automata (Kleene 1951), see also (Hopcroft and Ullman 1979; Rabin and Scott 1959). Regexes in the form as we know today have already been present in one of the pre-Unix implementations of the command-line text editor **qed** (Ritchie and Thompson 1970; the predecessor of the well-known **sed**).

Base R gives access to two different regex matching engines (via functions such as `gregexpr()` and `regexexec()`, see Table 5):

- ERE⁶ (*extended regular expressions* that conform to the **POSIX.2-1992** standard); used by default,
- PCRE⁷ (*Perl-compatible regular expressions*), in use if `perl=TRUE` is set.

Other matchers are implemented in the **ore** (Clayden 2019; via the **Onigmo** library) and **re2r** (Wenfeng 2020; **RE2**) packages.

Stringi, on the other hand, provides access to the regex engine implemented in **ICU**, which was inspired by Java’s **util.regex** in **JDK 1.4**. Their syntax is mostly compatible with that of **PCRE**, although certain advanced facets may not be supported (e.g., recursive patterns). On the other hand, **ICU** regexes fully conform to the Unicode Technical Standard #18 (Davis and Heninger 2021) and hence provide comprehensive support for Unicode.

It is worth noting that most programming languages as well as advanced text editors and IDEs (including **Kate**, **Eclipse**, **VSCode**, and **RStudio**) allow for finding or replacing patterns with regexes. Therefore, they should be amongst the instruments at every data scientist’s disposal. One general introduction to regexes is (Friedl 2006). The **ICU** flavour is summarised at <http://userguide.icu-project.org/strings/regexp>.

Below we provide a concise yet comprehensive introduction to the topic from the perspective of the **stringi** package users. This time we will use the pattern search routines whose names end

⁶Via the **TRE** library (<https://github.com/laurikari/tre/>).

⁷Via the **PCRE2** library (<https://www.pcre.org/>).

Option	Purpose
<code>case_insensitive</code> [regex flag (?i)]	logical; defaults to FALSE ; whether to enable (full) case-insensitive matching
<code>comments</code> [regex flag (?x)]	logical; defaults to FALSE ; whether to allow white spaces and comments within patterns
<code>dot_all</code> [regex flag (?s)]	logical; defaults to FALSE ; if set, “.” matches line terminators; otherwise its matching stops at a line end
<code>literal</code>	logical; defaults to FALSE ; whether to treat the entire pattern as a literal string; note that in most cases the code-pointwise string search facilities (<code>*_fixed()</code> functions described in Section 4) are faster
<code>multi_line</code> [regex flag (?m)]	logical; defaults to FALSE ; if set, “\$” and “^” recognise line terminators within a string; otherwise, they match only at start and end of the input
<code>unix_lines</code>	logical; defaults to FALSE ; when enabled, only the Unix line ending, i.e., U+000a, is honoured as a terminator by “.”, “\$”, and “^”
<code>uword</code> [regex flag (?w)]	logical; defaults to FALSE ; whether to use the Unicode definition of word boundaries (see Section 7.1), which are quite different from the traditional regex word boundaries
<code>error_on_unknown_escapes</code>	logical; defaults to FALSE ; whether unrecognised backslash-escaped characters trigger an error; by default, unknown backslash-escaped ASCII letters represent themselves
<code>time_limit</code>	integer; processing time limit for match operations in ~milliseconds (depends on the CPU speed); 0 for no limit (the default)
<code>stack_limit</code>	integer; maximal size, in bytes, of the heap storage available for the matcher’s backtracking stack; setting a limit is desirable if poorly written regexes are expected on input; 0 for no limit (the default)

Table 3: Options for the regular expressions search engine, see `stri_opts_regex()`.

with the `*_regex()` suffix. Apart from `stri_detect_regex()`, `stri_locate_all_regex()`, and so forth, in Section 5.4 we introduce `stri_match_all_regex()`, `stri_match_first_regex()`, and `stri_match_last_regex()`. Moreover, Table 3 lists the available options for the regex engine.

5.1. Matching individual characters

We shall begin by discussing different ways to define sets of characters. In this part, the length of all matching substrings will be quite easy to determine.

The following characters have special meaning to the regex engine:

. \ | () [{ } ^ \$ * + ?

Any regular expression that does not contain the above behaves like a fixed pattern:

```
R> x <- "spam, egg, spam, spam, bacon, and spam"
R> stri_count_regex(x, "spam")
```

```
[1] 4
```

However, this time the case insensitive mode fully supports Unicode matching⁸:

```
R> stri_detect_regex("groß", "GROSS", case_insensitive=TRUE)
```

```
[1] TRUE
```

If we wish to make a special character part of a regular expression – so that it is treated literally – we have to escape it with a backslash, “\”. Yet, the backslash itself has a special meaning to R, see `?Quotes`, therefore it needs to be preceded with another backslash.

```
R> stri_count_regex("spam...", "\\.") # "\\" is a way to input a single \
```

```
[1] 3
```

```
R> stri_count_regex("spam...", r"(\.)") # literal string - since R 4.0
```

```
[1] 3
```

In other words, the R string “\\.” is seen by the regex engine as “\.” and interpreted as the dot character (literally).

Matching any character. The (unescaped) dot, “.”, matches any character except the newline.

```
R> x <- "Ham, spam, jam, SPAM, eggs, and spam"
R> stri_extract_all_regex(x, ".am")
```

```
[[1]]
```

```
[1] "Ham" "pam" "jam" "pam"
```

```
R> stri_extract_all_regex(x, ".am", case_insensitive=TRUE)
```

⁸This does not mean, though, that it considers canonically equivalent strings as equal, see Section 6.2 for a discussion and a workaround.


```
[[1]]
[1] "Ham" "pam" "jam" "PAM" "pam"

R> stri_extract_all_regex(x, "..am", case_insensitive=TRUE)

[[1]]
[1] "spam" " jam" "SPAM" "spam"
```

The dot's insensitivity to the newline character is motivated by the need to maintain the compatibility with tools such as **grep** (when searching within text files in a line-by-line manner). This behaviour can be altered by setting the `dot_all` option to `TRUE`:

```
R> x <- "ham, spam, jam\namalgam"
R> stri_extract_all_regex(x, ".am")

[[1]]
[1] "ham" "pam" "jam" "gam"

R> stri_extract_all_regex(x, ".am", dot_all=TRUE)

[[1]]
[1] "ham" "pam" "jam" "\nam" "gam"
```

Defining character sets. Sets of characters can be introduced by enumerating their members between a pair of square brackets. For instance, “[abc]” denotes the set {a, b, c} – such a regular expression matches one (and only one) symbol from this set. Moreover, in:

```
R> stri_extract_all_regex(x, "[hjam]")

[[1]]
[1] "ham" "jam"
```

the “[hjam]” regex matches: “h” or “j”, followed by “a”, followed by “m”. In other words, “ham” and “jam” are the only two strings that are matched by this pattern (unless matching is done case-insensitively).

The following characters, if used within square brackets, may be treated non-literally:

\ [] ^ - &

Therefore, to include them as-is in a character set, the backslash-escape must be used. For example, “[\\[\\]]” matches the backslash or a square bracket.

Complementing sets. Including “^” after the opening square bracket denotes the set complement. Hence, “[^abc]” matches any character except “a”, “b”, and “c”.

```
R> x <- "Nobody expects the Spanish Inquisition!"
R> stri_extract_all_regex(x, "[^ ][^ ][^ ]")

[[1]]
[1] "Nob" "ody" "exp" "ect" "the" "Spa" "nis" "Inq" "uis" "iti" "on!"
```

The above matches any substring that consists of 3 non-spaces.

Defining Code Point Ranges. Each Unicode code point can be referenced by its unique numeric identifier, see Section 8.1 for more details. For instance, “a” is assigned code U+0061 and “z” is mapped to U+007A. In the pre-Unicode era (mostly with regards to the ASCII codes, \leq U+007F, representing English letters, decimal digits, some punctuation characters, and a few control characters), we were used to relying on specific code ranges; e.g., “[a-z]” denotes the set comprised of all characters with codes between U+0061 and U+007A, i.e., lowercase letters of the English (Latin) alphabet.

```
R> x <- "In 2020, I had fun once."
R> stri_extract_all_regex(x, "[a-z]")          # codes U+0061 - U+007A

[[1]]
[1] "n" "h" "a" "d" "f" "u" "n" "o" "n" "c" "e"

R> stri_extract_all_regex(x, "[0-9]")          # codes U+0030 - U+0039

[[1]]
[1] "2" "0" "2" "0"

R> stri_extract_all_regex(x, "[A-Za-z0-9]")    # union of 3 code ranges

[[1]]
[1] "I" "n" "2" "0" "2" "0" "I" "h" "a" "d" "f" "u" "n" "o" "n" "c" "e"
```

Nowadays, in the processing of text in natural languages, this notation should rather be avoided.

Using predefined character sets. Each code point is assigned a unique *general category*, which can be thought of as a character’s class, see (Whistler and Iancu 2021). Sets of characters from each category can be referred to, amongst others, by using the “\p{category}” syntax:

```
R> x <- "aąbßÆAĄB 你 123,.;'! \t-+=[ ]@+-” „ ² ³ ¼"
R> stri_extract_all_regex(x, "\\p{L}")        # letter (equivalently: [\p{L}])

[[1]]
[1] "a" "ą" "b" "ß" "Æ" "A" "Ą" "B" " " "你"

R> stri_extract_all_regex(x, "\\p{Ll}")      # lowercase letter

[[1]]
[1] "a" "ą" "b" "ß"

R> stri_extract_all_regex(x, "\\p{Lu}")      # uppercase letter
```

Characters' binary properties and scripts can also be referenced in a similar manner. Some other predefined classes include:

Moreover, e.g., the upper-cased “\P{category}” and “\W” is equivalent to “[^\p{category}]” and “[^\w]”, respectively, i.e., denote their complements.

In particular, in POSIX-like regex engines, “`[:punct:]`” stands for the character class corresponding to the `ispunct()` function in C (see `man 3 ispunct` on Unix-like systems). According to ISO/IEC 9899:1990 (ISO C90), `ispunct()` tests for any printing character except for the space or a character for which `isalnum()` is true.

In our case, **PCRE** yields:

```
R> x <- ",./|\\<>?:'\\"{ }-=_+()*&^%$€#@!`~x „"
R> stri_sub(x, gregexpr("[:punct:]", x, perl=TRUE)[[1]], length=1)

[1] "," "." "/" "|" "\\<" ">" "?" ";" ":" "' " "\"" "[" "]" "
[15] "{" "}" "-" "=" "_" "+" "(" ")" "*" "&" "^" "%" "$" "#"
[29] "@" "!" "`" "~"
```

```
R> stri_sub(x, gregexpr("[^[:punct:]]", x, perl=TRUE)[[1]], length=1)
```

```
[1] "€" "x" " " " „" " ” "
```

However, in a POSIX setting, the details of the characters' belongingness to particular classes depend on the current locale. Therefore, “[:punct:]”, in POSIX-like regex engines, is not portable.

ICU, on the other hand, gives:

```
R> stri_extract_all_regex(x, "[[:punct:]]") # equivalently: \p{P}

[[1]]
[1] "," "." "/" "\\<" "?" ";" ":" "' " "\"" "[" "]" "{" "}" "-"
[15] "_" "(" ")" "*" "&" "%" "#" "@" "!" " " " „" " ” "
```

```
R> stri_extract_all_regex(x, "[^[:punct:]]") # complement
```

```
[[1]]
[1] "|" "<" ">" "=" "+" "^" "$" "€" "`" "~" "x"
```

```
R> stri_extract_all_regex(x, "\\p{S}") # symbols
```

```
[[1]]
[1] "|" "<" ">" "=" "+" "^" "$" "€" "`" "~" "x"
```

We strongly recommend, wherever possible, the use of the portable “[\p{P}\p{S}]” as an alternative to the **PCRE** “[:punct:]”.

5.2. Alternating and grouping subexpressions

The alternation operator, “|”, allows us to match either its left or its right branch, for instance:

```
R> x <- "spam, egg, ham, jam, algae, and an amalgam of spam, all al dente"
R> stri_extract_all_regex(x, "spam|ham")
```

```
[[1]]
[1] "spam" "ham" "spam"
```

“|” has a very low precedence. Therefore, if we wish to introduce an alternative of *subexpressions*, we need to group them, e.g., between round brackets⁹:

```
R> stri_extract_all_regex(x, "(sp|h)am")
```

```
[[1]]
[1] "spam" "ham"  "spam"
```

Matching is always done left-to-right, on a first-come, first-served basis. Hence, if the left branch is a subset of the right one, the latter will never be matched, as in the example below:

```
R> stri_extract_all_regex(x, "(al/alga/algae)")
```

```
[[1]]
[1] "al" "al" "al" "al"
```

```
R> stri_extract_all_regex(x, "(algae/alga/al)")
```

```
[[1]]
[1] "algae" "alga"  "al"    "al"
```

Non-grouping parentheses. Some parenthesised subexpressions – those in which the opening bracket is followed by the question mark – have a distinct meaning. In particular, “(?#...)” denote free-format comments that are ignored by the regex parser:

```
R> stri_extract_all_regex(x,
+   "(?# match 'sp' or 'h')(sp|h)(?# and 'am')am|(?# or match 'egg')egg")
```

```
[[1]]
[1] "spam" "egg"  "ham"  "spam"
```

Nevertheless, constructing more sophisticated regexes by concatenating subfragments thereof may sometimes be more readable:

```
R> stri_extract_all_regex(x,
+   stri_paste(
+     "(sp|h)", # match either 'sp' or 'h'
+     "am",     # followed by 'am'
+     "|",      # ... or ...
+     "egg",    # just match 'egg'
+   ))
```

```
[[1]]
[1] "spam" "egg"  "ham"  "spam"
```

⁹Which have the side-effect of creating new capturing groups, see below for a discussion.

What is more, e.g., “(?i)” enables the `case_insensitive` mode.

```
R> stri_count_regex("Spam spam SPAMITY spAm", "(?i)spam")
```

```
[1] 4
```

For more regex flags, we kindly refer the reader to Table 3.

5.3. Quantifiers

Oftentimes, a variable number of instances of the same subexpression needs to be captured or its presence should be made optional. This can be achieved with the following quantifiers:

- “?” matches 0 or 1 times,
- “*” matches 0 or more times,
- “+” matches 1 or more times,
- “{n,m}” matches between `n` and `m` times,
- “{n,}” matches at least `n` times,
- “{n}” matches exactly `n` times.

These operators are applied to the preceding atoms. For example, “ba+” captures “ba”, “baa”, “baaa”, ... but not “b” alone.

By default, the quantifiers are *greedy* – they match the repeated subexpression as many times as possible. The “?” suffix (hence, “??”, “*?”, “+?”, and so forth) tries with as few occurrences as possible (to still get a match).

```
R> x <- "sp(AM)(maps)(SP)am"
R> stri_extract_all_regex(x,
+   c("\\(\\.+\\)",      # [[1]] greedy
+     "\\(\\.+?\\)",    # [[2]] lazy
+     "\\([\\^]+\\)"    # [[3]] greedy (but clever)
+ ))

[[1]]
[1] "(AM)(maps)(SP)"

[[2]]
[1] "(AM)" "(maps)" "(SP)"

[[3]]
[1] "(AM)" "(maps)" "(SP)"

R> stri_extract_first_regex("spamamamnomnomnomammmmmmmmm",
+   c("sp(am|nom)+",      "sp(am|nom)+?",
+     "sp(am|nom)+?m*",   "sp(am|nom)+?m+"))
```

```
[1] "spamamamnomnomnomam"      "spam"
[3] "spam"                      "spamamamnomnomnomammmmmmmmm"
```

Let us stress that the quantifier is applied to the subexpression that stands directly before it. Grouping parentheses can be used in case they are needed.

```
R> stri_extract_all_regex("12, 34.5, 678.901234, 37...629, ...",
+   c("\\d+\\.\\.\\d+",
+     "\\d+\\.\\.\\d+?",
+     "\\d+(\\.\\.\\d+)?"))
```

```
[[1]]
[1] "34.5"      "678.901234"
```

```
[[2]]
[1] "34.5"      "678.9"
```

```
[[3]]
[1] "12"          "34.5"          "678.901234" "37"          "629"
```

Performance notes. ICU, just like **PCRE**, uses a nondeterministic finite automaton-type algorithm. Hence, due to backtracking, some ill-defined regexes can lead to exponential matching times (e.g., “(a+)+b” applied on “aaaa...aaaaac”). If such patterns are expected, setting the `time_limit` or `stack_limit` option is recommended.

```
R> system.time(tryCatch({
+   stri_detect_regex("a" %s% 1000 %s% "c", "(a+)+b", time_limit=1e5)
+ }, error=function(e) cat("stopped.")))
```

stopped.

```
      user  system elapsed
31.345    0.000   31.345
```

Nevertheless, oftentimes such regexes can be naturally reformulated to fix the underlying issue. The **ICU** User Guide on Regular Expressions also recommends using *possessive quantifiers* (“?+”, “*+”, “++”, and so on), which match as many times as possible but, contrary to the plain-greedy ones, never backtrack when they happen to consume too much data.

See also the **re2r** (a wrapper around the **RE2** library; [Wenfeng 2020](#)) package’s documentation and the references therein for a discussion.

5.4. Capture groups and references thereto

It turns out that round-bracketed subexpressions carries one additional characteristic: they form the so-called *capture groups* that can be extracted separately or be referred to in other parts of the same regex.

Extracting capture group matches. The above is most evident when we use the versions of `stri_extract()` that are sensitive to the presence of capture groups: `stri_match_first_regex()`, `stri_match_last_regex()`, and `stri_match_all_regex()`.

```
R> x <- "name='Sir Launcelot', quest='Seek the Grail', colour='blue'"
R> stri_extract_all_regex(x, "\\w+= '(.*?)'")
```

```
[[1]]
[1] "name='Sir Launcelot'" "quest='Seek the Grail'"
[3] "colour='blue'"
```

```
R> stri_match_all_regex(x, "\\w+= '(.*?)'")
```

```
[[1]]
      [,1]          [,2]      [,3]
[1,] "name='Sir Launcelot'" "name"  "Sir Launcelot"
[2,] "quest='Seek the Grail'" "quest" "Seek the Grail"
[3,] "colour='blue'"      "colour" "blue"
```

In the latter example, we follow the convention introduced in **stringr**, where the findings are presented in a matrix form. The first column gives the complete matches, the second column stores the matches to the first capture group, and so forth.

If we just need the grouping part of “(...)”, i.e., without the capturing feature, “(?:...)” can be applied:

```
R> stri_match_all_regex(x, "(?:\\w+= '(.*?)'")
```

```
[[1]]
      [,1]          [,2]
[1,] "name='Sir Launcelot'" "Sir Launcelot"
[2,] "quest='Seek the Grail'" "Seek the Grail"
[3,] "colour='blue'"      "blue"
```

Replacing with capture group matches. Matches to particular capture groups can be recalled in replacement strings when using `stri_replace()`. Here, the match in its entirety is denoted with “\$0”, “\$1” stores whatever was caught by the first capture group, “\$2” is the match to the second capture group, etc. Moreover, “\\\$” gives the dollar-sign.

```
R> stri_replace_all_regex(x, "\\w+= '(.*?)'", "$2 is a $1")
```

```
[1] "Sir Launcelot is a name, Seek the Grail is a quest, blue is a colour"
```


Back-referencing. Matches to capture groups can also be part of the regexes themselves. For example, “\1” denotes whatever has been consumed by the first capture group.

Even though, in general, parsing HTML code with regexes is not recommended, let us consider the following examples:

```
R> x <- "<strong><em>spam</em></strong><code>eggs</code>"
R> stri_extract_all_regex(x, "<[a-z]+>.*?</[a-z]+>")

[[1]]
[1] "<strong><em>spam</em>" "<code>eggs</code>"

R> stri_extract_all_regex(x, "<([a-z]+)>.*?</\\1>") # \1 - back-reference

[[1]]
[1] "<strong><em>spam</em></strong>" "<code>eggs</code>"
```

The second regex guarantees that the match will include all characters between the opening `<tag>` and the *corresponding* (not: any) closing `</tag>`.

On a side note, currently **ICU** does not support the extraction of names of named capture groups, see however (Hocking 2019) for a discussion.

5.5. Anchoring

Lastly, let us mention the ways to match a pattern at a given abstract position within a string.

Matching at the beginning or end of a string. “^” and “\$” allow us to match, respectively, start and end of the string (or each line within a string, if the `multi_line` option is set to `TRUE`).

```
R> x <- c("spam egg", "bacon spam", "spam", "egg spam bacon")
R> stri_detect_regex(x, "spam") # 'spam' wherever

[1] TRUE TRUE TRUE TRUE

R> stri_detect_regex(x, "^spam") # begins with 'spam'

[1] TRUE FALSE TRUE FALSE

R> stri_detect_regex(x, "spam$") # ends with 'spam'

[1] FALSE TRUE TRUE FALSE

R> stri_detect_regex(x, "^spam$") # 'spam' only

[1] FALSE FALSE TRUE FALSE

R> stri_detect_regex(x, "spam$|^spam") # begins or ends with 'spam'

[1] TRUE TRUE TRUE FALSE
```

Matching at word boundaries. Furthermore, “\b” matches at a “word boundary“, e.g., near spaces, punctuation marks, or at the start/end of a string (i.e., wherever there is a transition between a word, “\w”, and a non-word character, “\W”, or vice versa).

In the two following examples we match all complete “words” that end with “am” (not just any string that includes “am”) and all stand-alone numbers¹⁰:

```
R> stri_extract_all_regex("spam, spams, jam, tramway", "\\b\\w*am\\b")
```

```
[[1]]
[1] "spam" "jam"
```

```
R> stri_extract_all_regex("12, 34.5, J23, 37.629cm", "\\b\\d+(\\.\\d+)?+\\b")
```

```
[[1]]
[1] "12"    "34.5"
```

Looking behind and ahead. There are also ways to guarantee that a pattern occurrence begins or ends with a match to some subexpression: “(?<=...)...” is the so-called *look-behind*, whereas “...(?=...)” denotes the *look-ahead*. Moreover, “(?<!...)...” and “...(?!...)” are their negated (“negative look-behind/ahead”) versions.

```
R> x <- "I like spam, spam, eggs, and spam."
R> stri_extract_all_regex(x, "\\w+(?=[,.]") # word that ends with ',' or '.'
```

```
[[1]]
[1] "spam" "spam" "eggs" "spam"
```

```
R> stri_extract_all_regex(x, "\\w++(?![,.]") # neither ends with ',' nor '.'
```

```
[[1]]
[1] "I"    "like" "and"
```

6. String collation

Historically, code-pointwise comparison had been used in most string comparison activities, especially if strings in ASCII (i.e., English) were involved. However, nowadays this does not necessarily constitute the most suitable approach to the processing of natural-language texts. In particular, a code-pointwise matching neither takes accented and conjoined letters nor ignorable punctuation and case into account.

The **ICU Collation Service**¹¹ provides the basis for string comparison activities such as string sorting and searching, or determining if two strings are equivalent. This time, though, due to its conformance to the Unicode Collation Algorithm (Davis, Whistler, and Scherer 2021),

¹⁰This regex is for didactic purposes only.

¹¹See the ICU User Guide on Collation, <http://userguide.icu-project.org/collation>.

we may expect that the generated results will meet the requirements of the culturally correct natural language processing in any *locale*.

6.1. Locales

String collation is amongst many locale-sensitive operations available in **stringi**. Before proceeding any further, we should first discuss how we can parameterise the **ICU** services so as to deliver the results that reflect the expectations of a specific user community, such as the speakers of different languages and their various regional variants.

Specifying locales. A locale specifier¹² is of the form "Language", "Language_Country", or "Language_Country_Variant", where:

- **Language** is, most frequently, a two- or three-letter code that conforms to the ISO-639-1 or ISO-639-2 standard, respectively; e.g., "en" or "eng" for English, "es" or "spa" for Spanish, "zh" or "zho" for Chinese, and "mas" for Masai (which lacks the corresponding two-letter code); however, more specific language identifiers may also be available, e.g., "zh_Hans" for Simplified- and "zh_Hant" for Traditional-Chinese or "sr_Cyrl" for Cyrillic- and "sr_Latn" Latin-Serbian;
- **Country** is a two-letter code following the ISO-3166 standard that enables different language conventions within the same language; e.g., the US-English ("en_US") and Australian-English ("en_AU") not only observe some differences in spelling and vocabulary, but also in the units of measurement;
- **Variant** is an identifier indicating a preference towards some convention within the same country; e.g., "de_DE_PREEURO" formats currency values using the pre-2002 Deutsche Mark (DEM).

Moreover, following the “@” symbol, semicolon-separated “key=value” pairs can be appended to the locale specifier, in order to customise some locale-sensitive services even further (see below for an example using “@collation=phonebook” and Section 7.5 for “@calendar=hebrew”, amongst others).

Listing locales. To list the available locale identifiers, we call `stri_locale_list()`.

```
R> length(stri_locale_list())      # number of available locales
```

```
[1] 782
```

```
R> sample(stri_locale_list(), 5)  # 5 random ones
```

```
[1] "nl_BQ" "pt_CH" "fi"      "en_PN" "en_IL"
```

¹²Locale specifiers in **ICU** are platform-independent. This is not the case for their base-R counterparts, see `help("locales")`, e.g., we have "Polish_Poland" on Windows vs. "pl_PL" on Linux.

Querying for locale-specific services. The availability of locale-specific services can only be determined during the very request for a particular resource¹³. It may depend on the **ICU** library version actually in use as well as the way the **ICU** Data Library (**icudt**) has been packaged. Therefore, for maximum portability, it is best to rely on the **ICU** library bundle that is shipped with **stringi**. This is the case on Windows and OS X, whose users typically download the pre-compiled versions of the package from CRAN. However, on various flavours of GNU/Linux and other Unix-based systems, the system **ICU** is used more eagerly¹⁴. To force building **ICU** from sources, we may call:

```
R> install.packages("stringi", configure.args="--disable-pkg-config")
```

Overall, should a requested service be unavailable in a given locale, the best possible match is returned.

Default locale. Each locale-sensitive operation in **stringi** selects the *current default locale* if no locale has been explicitly requested, i.e., when a function’s **locale** argument (see Table 4) is left alone in its “NULL” state. The default locale is initially set to match the system locale on the current platform, and may be changed with **stri_locale_set()**, e.g., in the very rare case of improper automatic locale detection.

```
R> stri_locale_get()
```

```
[1] "en_AU"
```

6.2. Testing string equivalence

In Unicode, some characters may have multiple representations. For instance, “LATIN SMALL LETTER A WITH OGONEK” (“ą”) can be stored as a single code point U+0105 or as a sequence that is comprised of the letter “LATIN SMALL LETTER A”, U+0061, and the “COMBINING OGONEK”, U+0328 (when rendered properly, they should appear as if they were identical glyphs). This is an example of *canonical equivalence* of strings.

Testing for the Unicode equivalence between strings can be performed by calling **%s==%** and, more generally, **stri_cmp_equiv()**, or their negated versions, **%s!=%** and **stri_cmp_nequiv()**.

```
R> "a\u0328" %s==% "ą" # a, ogonek == a with ogonek
```

```
[1] TRUE
```

```
R> stri_cmp_equiv("a\u0328", "ą") # the same
```

```
[1] TRUE
```

¹³For more details, see the **ICU** User Guide on Locales, <http://userguide.icu-project.org/locale>.

¹⁴See, e.g., software packages **libicu-dev** on Debian/Ubuntu or **libicu-devel** on RHEL/Fedora/OpenSUSE. For more details regarding the configure/build process of **stringi**, refer to the **INSTALL** file.

There are also functions for indicating and removing duplicated elements in a character vector:

```
R> x <- c("Gagolewski", "Gagolewski", "Ga\u0328golewski")
R> stri_unique(x)

[1] "Gagolewski" "Gagolewski"

R> stri_duplicated(x)

[1] FALSE FALSE  TRUE

R> stri_duplicated(x, from_last=TRUE)

[1]  TRUE FALSE FALSE

R> stri_duplicated_any(x) # index of the first non-unique element

[1] 3
```

6.3. Linear ordering of strings

Operators such that `%s<%`, `%s<=%`, etc., and the corresponding functions `stri_cmp_lt()` (“less than”), `stri_cmp_le()` (“less than or equal”), etc., implement locale-sensitive linear orderings of strings. Moreover, `stri_sort()` returns the lexicographically-sorted version of a given input vector and `stri_order()` yields the corresponding (stable) ordering permutation.

```
R> "chaotic" %s<% "hard" # current default locale (here: en_AU)

[1] TRUE

R> stri_cmp_lt("chłodny", "hardy", locale="pl_PL") # Polish

[1] TRUE

R> stri_cmp_lt("chladný", "hladný", locale="sk_SK") # Slovak

[1] FALSE

R> stri_cmp("chladný", "hladný", locale="sk_SK") # -1,0,1 encode <,<=,>

[1] 1
```

Note that the locale-aware comparison might be context-sensitive and goes beyond the simple code-pointwise comparison. In the example above, a *contraction* occurred: in the Slovak language, two code points “ch” are treated as a single entity and are sorted after “h”:

```
R> stri_sort(c("chłodny", "hardy", "cichy", "cenny"), locale="pl_PL")
```

```
[1] "cenny"    "chłodny" "cichy"    "hardy"
```

```
R> stri_sort(c("cudný", "chladný", "hladný", "čudný"), locale="sk_SK")
```

```
[1] "cudný"    "čudný"    "hladný"    "chladný"
```

An opposite situation is called an *expansion*:

```
R> german_k_words <- c("können", "kondensieren", "kochen", "korrelieren")
```

```
R> stri_sort(german_k_words, locale="de_DE")
```

```
[1] "kochen"      "kondensieren" "können"      "korrelieren"
```

```
R> stri_sort(german_k_words, locale="de_DE@collation=phonebook")
```

```
[1] "kochen"      "können"      "kondensieren" "korrelieren"
```

In the latter example, where we use the German phone-book order, "ö" is treated as "oe".

6.4. Collator options

Table 4 lists the options that can be passed to `stri_opts_coll()` via “...” in all the functions that rely on the ICU Collator. Below we would like to attract the reader’s attention to some of them.

Collation strength. The Unicode Collation Algorithm (Davis *et al.* 2021) can go beyond simple canonical equivalence: they allow us to treat some other (depending on the context) differences as *negligible* too.

The `strength` option controls the Collator’s “attention to detail”. For instance, it can be used to make the ligature “ff” (U+FB00) compare equal to the two-letter sequence “ff”:

```
R> stri_cmp_equiv("\ufb00", "ff")
```

```
[1] FALSE
```

```
R> stri_cmp_equiv("\ufb00", "ff", strength=2)
```

```
[1] TRUE
```

Option	Purpose
<code>locale</code>	a string specifying the locale to use; <code>NULL</code> (default) or <code>""</code> for the current default locale as indicated by <code>stri_locale_get()</code>
<code>strength</code>	an integer in $\{1, 2, 3, 4\}$ defining collation strength; 1 for the most permissive collation rules, 4 for the strictest ones; defaults to 3
<code>uppercase_first</code>	logical; <code>NA</code> (default) orders upper and lower case letters in accordance to their tertiary weights, <code>TRUE</code> forces upper case letters to sort before lower case letters, <code>FALSE</code> does the opposite
<code>numeric</code>	logical; if <code>TRUE</code> , a collation key for the numeric value of substrings of digits is generated; this is a way to make <code>"100"</code> ordered after <code>"2"</code> ; defaults to <code>FALSE</code>
<code>case_level</code>	logical; if <code>TRUE</code> , an extra case level (positioned before the third level) is generated; defaults to <code>FALSE</code>
<code>normalisation</code>	logical; if <code>TRUE</code> , then an incremental check is performed to see whether input data are in the FCD (“fast C or D”) form; if data are not in the FCD form, the incremental NFD normalisation is performed, see Section 8.4; defaults to <code>FALSE</code>
<code>alternate_shifted</code>	logical; if <code>FALSE</code> (default), all code points with non-ignorable primary weights are handled in the same way; <code>TRUE</code> causes the code points with primary weights that not above the variable top value to be ignored on the primary level and moved to the quaternary level; this can be used to, e.g., ignore punctuation, see the examples provided
<code>french</code>	logical; <code>TRUE</code> results in secondary weights being considered backwards, i.e., ordering according to the last accent difference – nowadays only used in Canadian French; defaults to <code>FALSE</code>

Table 4: Options for the ICU Collator that can be passed to `stri_opts_collator()`.

Generally, four (nested) levels of inter-string differences can be distinguished:

1. A primary difference – the strongest one – occurs where there is a mismatch between base characters (e.g., `"a"` vs. `"b"`).
2. Some character accents can be considered a secondary difference in many languages. However, in other ones, an accented letter is considered a unique letter.
3. Distinguishing between upper- and lower case typically happens on the tertiary level, see, however, the `case_level` option.
4. If `alternate_shifted` is `TRUE`, differences in punctuation can be determined at the quaternary level. This is also meaningful in the processing of Hiragana text.

Ignoring case. Note what follows:

```
R> x <- c("gro\u00df", "gross", "GROSS", "Gro\u00df", "Gross")
R> stri_unique(x, strength=1) # \beta == ss, case insensitive

[1] "gro\u00df"

R> stri_unique(x, strength=1, case_level=TRUE) # \beta == ss, case sensitive

[1] "gro\u00df" "GROSS" "Gro\u00df"

R> stri_unique(x, strength=2) # \beta != ss, case insensitive

[1] "gro\u00df" "gross"
```

Ignoring some punctuation. Here are some effects of changing the `alternate_shifted` option:

```
R> x <- c("code point", "code-point", "codepoint", "CODE POINT", "CodePoint")
R> stri_unique(x, alternate_shifted=TRUE) # strength=3

[1] "code point" "CODE POINT" "CodePoint"

R> stri_unique(x, alternate_shifted=TRUE, strength=2)

[1] "code point"

R> stri_unique(x, strength=2)

[1] "code point" "code-point" "codepoint"
```

Backward secondary sorting. The French Canadian Sorting Standard CAN/CSA Z243.4.1 (historically this had been the default for all French locales) requires the word ordering with respect to the last accent difference. Such a behaviour can be applied either by setting the French-Canadian locale or by passing the `french=TRUE` option to the Collator.

```
R> stri_sort(c("cote", "c\u00f4te", "cot\u00e9", "c\u00f4t\u00e9"), locale="fr_FR")

[1] "cote" "cot\u00e9" "c\u00f4te" "c\u00f4t\u00e9"

R> stri_sort(c("cote", "c\u00f4te", "cot\u00e9", "c\u00f4t\u00e9"), locale="fr_CA") # french=TRUE

[1] "cote" "c\u00f4te" "cot\u00e9" "c\u00f4t\u00e9"
```


Sorting numerals. Moreover, let us note the effect of setting the `numeric` option on the sorting of strings that involves numbers:

```
R> stri_sort(c("a1", "a2", "a11", "a10", "a100")) # lexicographic order
```

```
[1] "a1"    "a10"   "a100"  "a11"   "a2"
```

```
R> stri_sort(c("a1", "a2", "a11", "a10", "a100"), numeric=TRUE)
```

```
[1] "a1"    "a2"    "a10"   "a11"   "a100"
```

A note on compatibility equivalence. In Section 8.4 we describe different ways to normalise canonically equivalent code point sequences so that they are represented by the same code points, which can account for some negligible differences (as in the “a with ogonek” example above).

Apart from ignoring punctuation and case, the Unicode Standard Annex #15 (Davis and Whistler 2021) also discusses the so-called *compatibility* equivalence of strings. This is a looser form of similarity; it is observed where there is the same *abstract* content, yet displayed by means of different glyphs, for instance “¼” (U+00BC) vs. “1/4” or “ℝ” vs. “R”. In the latter case, whether these should be treated as equal, depends on the context (e.g., this can be the set of real numbers vs. one’s favourite programming language). Compatibility decompositions (NFKC, NFKD) mentioned in Section 8.4 or other types of transliteration can be used to normalise strings so that such differences are not accounted for.

Also, for “fuzzy” matching of strings, the `stringdist` package (van der Loo 2014) might be helpful.

6.5. Searching for fixed strings revisited

The ICU Collator can also be utilised where there is a need to locate the occurrences of simple textual patterns. All the string search functions described in Section 4 have their `*_coll()`-suffixed equivalents. Despite being slower than their `*_fixed()` counterparts, they are more appropriate in NLP activities.

```
R> stri_detect_coll("Er ist so groß.", "GROSS", strength=1, locale="de_AT")
```

```
[1] TRUE
```

```
R> stri_detect_coll("On je chladný", "chladny", strength=1, locale="sk_SK")
```

```
[1] TRUE
```

7. Other operations

In the sequel, we cover the functions that deal with text boundaries’ detection, random string generation, date/time formatting and parsing, amongst others.

7.1. Analysing text boundaries

Text boundary analysis aims at locating linguistic delimiters for the purpose of word-wrapping of text, counting characters or words, locating particular text units (e.g., the 3rd sentence), etc.

Generally, text boundary analysis is a locale-sensitive operation, see (Davis and Chapman 2021). For example, in Japanese and Chinese, spaces are not used for the separating of words – a line break can occur even in the middle of a word. Nevertheless, these languages have punctuation and diacritical marks that cannot start or end a line, so this must also be taken into account.

The **ICU Break Iterator**¹⁵ comes in four flavours (see the `type` option in `stri_opts_brkiter()`): `character`, `work`, `line_break`, and `sentence`.

We have access to functions such as `stri_count_boundaries()`, `stri_split_boundaries()`, `stri_extract_*_boundaries()`, and `stri_locate_*_boundaries()`, as well as their specialised versions: `stri_count_words()`, `stri_extract_*_words()`, and `stri_split_lines()`, amongst others. For example:

```
R> x <- "The\u00a0above-mentioned    features are useful. " %s+%
+      "My hovercraft is full of eels, eggs, and spam."
R> stri_count_boundaries(x, type="sentence") # number of sentences

[1] 2

R> stri_count_boundaries(x, type="word") # number of word boundaries

[1] 33

R> stri_count_words(x) # number of words themselves

[1] 15

R> stri_extract_all_words(x)

[[1]]
 [1] "The"      "above"    "mentioned" "features"  "are"
 [6] "useful"   "My"       "hovercraft" "is"        "full"
[11] "of"       "eels"     "eggs"      "and"       "spam"
```

7.2. Trimming, padding, and other formatting

The following functions can be useful when pretty-printing character strings or text on the console, dynamically generating reports (e.g., with `Sweave()` or **knitr**; see Xie 2015), or creating text files (e.g., with `stri_write_lines()`; see Section 8.3).

¹⁵See the **ICU User Guide on Boundary Analysis**, <http://userguide.icu-project.org/boundaryanalysis>.

Padding. `stri_pad()` pads strings with some character so that they reach the desired lengths. This can be used to centre, left-, or right-align a message when printed with, e.g., `cat()`.

```
R> cat(stri_pad("spam", width=77, side="left"))
```

spam

```
R> cat(stri_pad("SPAMITY SPAM", width=77, side="both", pad="."))
```

```
.....SPAMITY SPAM.....
```

Trimming. A dual operation is that of trimming from the left or right side of strings:

```
R> x <- "      spam, eggs, and lovely spam.\n"
```

```
R> stri_trim(x) # side="both"
```

```
[1] "spam, eggs, and lovely spam."
```

```
R> stri_trim(x, pattern="[^\n\\p{Z}\\p{P}\\p{S}]")
```

```
[1] "spam, eggs, and lovely spam"
```

Word wrapping. The `stri_wrap()` function splits each (possibly long) string in a character vector into chunks of at most a given width or length. By default, the dynamic word wrap algorithm ([Knuth and Plass 1981](#)) that minimises the raggedness of the formatted text is used. However, there is also an option (`cost_exponent=0`) to use the greedy alignment, for compatibility with the built-in `strwrap()`.

```
R> x <- stri_rand_lipsum(1) # random text paragraph
```

```
R> cat(stri_wrap(x, width=60, indent=24, exdent=20, prefix="> "), sep="\n")
```

```
>               Lorem ipsum dolor sit amet, quis
>               donec pretium auctor, quis id. Mauris
>               rhoncus donec amet egestas sagittis
>               ipsum per. Sed, sociis amet. Aliquam
>               fusce dictumst sed vehicula ultrices
>               arcu. Eros, netus et. Amet amet mi
>               vestibulum vitae dapibus ut felis.
>               Magnis in vestibulum egestas massa
>               curabitur a ut, eget in in facilisis.
>               Etiam odio fermentum sit ante
>               ridiculus sit elit. Sapien torquent
>               fermentum tortor gravida ornare sapien
>               consequat et sem turpis. Hac vel lacus
>               habitasse et id non. Metus habitasse
>               sed lacinia nibh ex metus. Amet nam
>               vestibulum ornare tincidunt massa sed
>               ullamcorper.
```

Note that by default splitting is performed at line breaks (compare Section 7.1).

Applying string templates. The binary operator `%s$%` provides access to the built-in `sprintf()` in a way similar to Python’s `%` overloaded for objects of type `str`.

```
R> "value='%d'" %s$% 3                # equivalently: "value='%d'" %s$% list(3)
```

```
[1] "value='3'"
```

```
R> "%s='%d'" %s$% list("value", 1:3)
```

```
[1] "value='1'" "value='2'" "value='3'"
```

7.3. Generating random strings

Apart from `stri_rand_lipsum()`, which produces random-ish text paragraphs (“placeholders” for real text), we have access to a function that generates sequences of characters uniformly sampled (with replacement) from a given set.

```
R> stri_rand_strings(5, 8, "[actg]")
```

```
[1] "ctcttagt" "gctcggat" "aacttggt" "ggggcatt" "gtactaca"
```

```
R> stri_rand_strings(5, 2:6, "[A-Za-z]")
```

```
[1] "HV"      "VTH"     "HMYN"    "sCWpG"   "dKGnuT"
```

```
R> stri_rand_strings(1, 8, "[\\p{script=Katakana}&\\p{L}]")
```

```
[1] " ヲケメハレツテ"
```

See Section 5.1 for different ways to specify character sets.

7.4. Transliterating

Transliteration, in its broad sense, deals with the substitution of characters or their groups for different ones, according to some well-defined rules. It may be useful, amongst others, when “normalising” pieces of strings or identifiers so that they can be more easily compared with each other.

Case mapping. Mapping to upper, lower, or title case is a language- and context-sensitive operation that can change the total number of code points in a string.

```
R> stri_trans_toupper("groß")

[1] "GROSS"

R> stri_trans_tolower("İİ", locale = "tr_TR")           # Turkish

[1] "ii"

R> stri_trans_totitle("ijsvrij yoghurt", locale = "nl_NL") # Dutch

[1] "IJsvrij Yoghurt"
```

Mapping between specific characters. If a fast 1-to-1 exchange of characters is required, we can call:

```
R> stri_trans_char("GATAAATCTGGTCTTATTTCC", "ACGT", "tgca")

[1] "ctatttagaccagaataaagg"
```

Here, “A”, “C”, “G”, and “T” is replaced with “t”, “g”, “c”, and “a”, respectively.

General transforms. The `stri_stats_general()` function provides access to a wide range of text transforms defined by **ICU**¹⁶, whose catalogue can be accessed by calling `stri_trans_list()`.

```
R> sample(stri_trans_list(), 9) # a few random entries

[1] "Kannada-Gurmukhi"      "Deva-Taml"
[3] "Macedonian-Latin/BGN"  "Any-bg_Latn/BGN"
[5] "Any-fa_FONIPA"         "Any-CanadianAboriginal"
[7] "cs-chr"                "Latn-Taml"
[9] "Gujr-Arab"
```

Some examples:

```
R> stri_trans_general("groß© zóĺć La Niña köszönöm", "upper; latin-ascii")

[1] "GROSS(C) ZOLC LA NINA KOSZONOM"

R> stri_trans_general("Let's go... -- she said.", "any-publishing")
```

¹⁶See the **ICU** User Guide on General Transforms, <http://userguide.icu-project.org/transforms/general>.

```
[1] "Let' s go...— she said."
```

7.5. Parsing and formatting date and time

In base R, dealing with temporal data in languages different than the current locale is somewhat difficult. For instance, many will find the task of parsing the following Polish date problematic:

```
R> x <- "27 sierpnia 2020 r., godz. 17:17:32"
```

stringi connects to the **ICU** date and time services so that this becomes an easy exercise:

```
R> stri_datetime_parse(x, "dd MMMM yyyy 'r., godz.' HH:mm:ss",
+   locale="pl_PL", tz="Europe/Warsaw")
```

```
[1] "2020-08-27 17:17:32 CEST"
```

This function returns an object of class `POSIXct`, for compatibility with base R. Note, however, that **ICU** uses its own format patterns¹⁷. For convenience, `strftime()`- and `strptime()`-like templates can be converted with `stri_datetime_fstr()`:

```
R> stri_datetime_parse(x,
+   stri_datetime_fstr("%d %B %Y r., godz. %H:%M:%S"),
+   locale="pl_PL", tz="Europe/Warsaw")
```

```
[1] "2020-08-27 17:17:32 CEST"
```

Some more examples:

```
R> stri_datetime_format(stri_datetime_now(), # current date and time
+   "datetime_full",                        # full format
+   locale="de_AT", tz="Europe/Vienna")
```

```
[1] "Montag, 15. Februar 2021 um 01:06:21 Mitteleuropäische Normalzeit"
```

```
R> stri_datetime_format(
+   stri_datetime_add(stri_datetime_now(), 1, "day"), # add 1 day to 'now'
+   "datetime_relative_long",                        # full format, relative to 'now'
+   locale="en_NZ", tz="NZ")
```

```
[1] "tomorrow at 1:06:21 pm NZDT"
```

¹⁷See the **ICU** User Guide on Formatting Dates and Times, <http://userguide.icu-project.org/formatparse/datetime>.

```
R> stri_datetime_format(
+   stri_datetime_create(2020, 1:12, 1),    # vectorised w.r.t. all arguments
+   "date_long",                           # date only
+   locale="@calendar=hebrew")             # English locale, Hebrew calendar

[1] "4 Tevet 5780"    "6 Shevat 5780"    "5 Adar 5780"      "7 Nisan 5780"
[5] "7 Iyar 5780"     "9 Sivan 5780"     "9 Tamuz 5780"     "11 Av 5780"
[9] "12 Elul 5780"    "13 Tishri 5781"   "14 Heshvan 5781"  "15 Kislev 5781"

R> stri_datetime_format(
+   stri_datetime_create(2020, c(2, 8), c(4, 7)),
+   "date_full",
+   locale="ja_JP@calendar=japanese")      # Japanese locale and calendar

[1] " 令和 2 年 2 月 4 日火曜日" " 令和 2 年 8 月 7 日金曜日"
```

8. Input and output

This section deals with some more advanced topics related to the interoperability between different platforms. In particular, we discuss how to assure that data read from input connections are interpreted in the correct manner.

8.1. Dealing with Unicode code points

The Unicode Standard (as well as the Universal Coded Character Set, i.e., ISO/IEC 10646) currently defines over 140,000 abstract characters together with their corresponding *code points* – integers between 0 and 1,114,111 (or 0000_{16} and $10FFFF_{16}$ in hexadecimal notation, see <https://www.unicode.org/charts/>). In particular, here are the counts of the code points in a few popular categories (compare Section 5.1), such as letters, numbers, and the like.

```
R> z <- c("\\p{L}", "\\p{Ll}", "\\p{Lu}", "\\p{N}", "\\p{P}", "\\p{S}",
+   "\\w", "\\d", "\\s")
R> structure(stri_count_regex(stri_enc_fromutf32(
+   setdiff(1:0x10ffff, c(0xd800:0xf8ff))), z), names=z)

\\p{L} \\p{Ll} \\p{Lu} \\p{N} \\p{P} \\p{S}   \\w   \\d   \\s
131241  2155   1791   1781   798   7564 134564  650   25
```

Yet, most of the code points are still unallocated – the Unicode standard receives some updates from time to time, e.g., the most recent versions were supplemented with over 1,000 emojis. The first 255 code points are identical to the ones defined by ISO/IEC 8859-1 (ISO Latin-1; “Western European”), which itself extends US-ASCII (codes $\leq 127 = 7F_{16}$). For instance, the code point that we are used to denoting as U+007A (the “U+” prefix is followed by a sequence of hexadecimal digits; $7A_{16}$ corresponds to decimal 122) encodes the lower case letter “z”. To input such a code point in R, we write:

```
R> "\u007A" # or "\U0000007A"
```

```
[1] "z"
```

For communicating with **ICU** and other libraries, we may need to escape a given string, for example, as follows (recall that to input a backslash in R, we must precede it with another backslash).

```
R> x <- "z\u00df\u4f60\u597d"
R> stri_escape_unicode(x)
```

```
[1] "z\\u00df\\u4f60\\u597d"
```

```
R> stri_trans_general(x, "any-hex")
```

```
[1] "\\u007A\\u00DF\\u4F60\\u597D"
```

```
R> stri_trans_general(x, "[^\u0000-\u007f] any-hex") # except ASCII
```

```
[1] "z\\u00DF\\u4F60\\u597D"
```

```
R> stri_trans_general(x, "[^\u0000-\u007f] any-hex/xml")
```

```
[1] "z&#xDF;&#x4F60;&#x597D;"
```

It is worth noting that despite the fact that some output devices might be unable to display certain code points correctly (due to, e.g., missing fonts), the correctness of their processing with **stringi** is still guaranteed by **ICU**. Here is an example of an incorrect *presentation* of an emoji, generated by a malconfigured $\text{Xe}_{\text{L}}\text{A}_{\text{T}}\text{E}_{\text{X}}$ engine:

```
R> "\U001F600" # the grinning face emoji, (:
```

```
- font unavailable
```

```
[1] " "
```

Nevertheless, the programmatic handling of such a code point is unaffected:

```
R> stri_trans_general("\U001F600", "any-name") # query the character database
```

```
[1] "\\N{GRINNING FACE}"
```

8.2. Character encodings

When storing strings in RAM or on the disk, we need to decide upon the actual way of representing the code points as sequences of bytes. The two most popular *encodings* in the Unicode family are UTF-8 and UTF-16:


```
R> x <- "abz0aß 你好!"
R> stri_encode(x, to="UTF-8", to_raw=TRUE)[[1]]

[1] 61 62 7a 30 c4 85 c3 9f e4 bd a0 e5 a5 bd 21

R> stri_encode(x, to="UTF-16LE", to_raw=TRUE)[[1]]

[1] 61 00 62 00 7a 00 30 00 05 01 df 00 60 4f 7d 59 21 00
```

R's current platform-default encoding, which we shall refer to as the *native* encoding, is defined via the `LC_CTYPE` locale category in `Sys.getlocale()`. This is the representation assumed, e.g., when reading data from the standard input or files (e.g., when `scan()` is called). For instance, Central European versions of Windows will assume the “windows-1250” code page. OS X as well as most Linux boxes work with UTF-8 by default.

All strings in R have an associated encoding mark which can be read by calling `Encoding()` or, more conveniently, `stri_enc_mark()`. Most importantly, strings in ASCII, ISO-8859-1 (“latin1”), UTF-8, and the native encoding can coexist. Whenever a non-Unicode string is passed to a **stringi** function, it is silently converted to UTF-8 or UTF-16, depending on the requested operation (some **ICU** services are only available for UTF-16 data). Over the years, this has proven a robust, efficient, and maximally portable design choice – Unicode can be thought of as a superset of every other encoding. Moreover, in order to guarantee the correctness and high performance of the string processing pipelines, **stringi** always¹⁸ outputs UTF-8 data.

8.3. Reading and writing text files and converting between encodings

According to a report by W3Techs¹⁹, as of 2021-02-15, 96.3% of websites use UTF-8. Nevertheless, other encodings can still be encountered.

Reading and writing text files. If we know the encoding of a text file in advance, `stri_read_lines()` can be used to read the data in a manner similar to the built-in `readLines()` function (but with a much easier access to encoding conversion):

```
R> x <- stri_read_lines("ES_latin1.txt", encoding="ISO-8859-1")
R> head(x) # now x is in UTF-8

[1] "LOS CONSEJOS DE UN PADRE"
[2] ""
[3] ""
[4] "El León, el rey de las selvas, agonizaba en el hueco de su caverna...."
[5] ""
[6] "Á su lado estaba su hijo, el nuevo león, el rey futuro de todos los"
```

We can call `stri_write_lines()` to write the contents of a character vector to a file (each string will constitute a separate text line), with any output encoding.

¹⁸With a few obvious exceptions, such as `stri_encode()`.

¹⁹See https://w3techs.com/technologies/cross/character_encoding/ranking.

Detecting encoding. However, if a file's encoding is not known in advance, there are a certain functions that can aid in encoding detection. First, we can read the resource in form of a raw-type vector:

```
R> x <- stri_read_raw("ES_latin1.txt")
R> head(x) # vector of type raw

[1] 4c 4f 53 20 43 4f
```

Then, to guess the encoding, we can call, e.g.:

```
R> stri_enc_isascii(x)

[1] FALSE

R> stri_enc_isutf8(x) # false positives are possible

[1] FALSE
```

Alternatively, we can use:

```
R> stri_enc_detect(x) # based on heuristics

[[1]]
      Encoding Language Confidence
1 ISO-8859-1      es      0.74
2 ISO-8859-2      ro      0.32
3 ISO-8859-9      tr      0.13
4 UTF-16BE                0.10
5 UTF-16LE                0.10
```

Nevertheless, encoding detection is an operation that relies on heuristics, therefore there is a chance that the output might be imprecise or even misleading.

Converting encodings. Knowing the desired source and destination encoding precisely, `stri_encode()` can be called to perform the conversion. Contrary to the built-in `iconv()`, which relies on different underlying libraries, the current function is portable across operating systems.

```
R> y <- stri_encode(x, from="ISO-8859-1", to="UTF-8")
R> # split into text lines
R> tail(stri_split_lines1(y)) # spoiler alert!

[1] "El mono saltó sobre el perro, y en él se montó imitando al hombre;"
[2] "caballo perruno y caballero cuadrumano, salieron corriendo por el"
[3] "bosque."
[4] ""
[5] "El águila se remontó, diciendo:--El hombre mató al león; hay que subir"
[6] "mucho para que no me alcance; ¿quién sabe si algún día me alcanzará?"
```

`stri_enc_list()` provides a list of supported encodings and their aliases in many different forms. Encoding specifiers are normalised automatically, e.g., "utf8" is a synonym for "UTF-8".

8.4. Normalising strings

In Section 6.2 we have provided some examples of canonically equivalent strings whose code point representation was different. Unicode normalisation forms C (Canonical composition, NFC) and D (Canonical decomposition, NFD) can be applied so that they will compare equal using bitwise matching (Davis and Whistler 2021).

```
R> x <- "a\u0328 a" # a, combining ogonek, space, a with ogonek
R> stri_enc_toutf32(x)[[1]] # code points as decimals
```

```
[1] 97 808 32 261
```

```
R> stri_enc_toutf32(stri_trans_nfc(x))[[1]]
```

```
[1] 261 32 261
```

```
R> stri_enc_toutf32(stri_trans_nfd(x))[[1]]
```

```
[1] 97 808 32 97 808
```

It might be a good idea to always normalise all the strings read from external sources (files, URLs) with NFC.

Compatibility composition and decomposition normalisation forms (NFKC and NFKD, respectively) are also available if the removal of the formatting distinctions (font variants, subscripts, superscripts, etc.) is desired:

```
R> stri_trans_nfkd("r²ˆ")
```

```
[1] "r2{"
```

9. Closing remarks

We have reviewed the key design principles and facilities available in **stringi**, including numerous operations that help implement and optimise data processing pipelines. Let us stress again that many of the package features are provided by **ICU**, which is a platform-independent project. Hence, information presented above might be of interest to statisticians and data scientists employing different programming languages as well.

Portability issues in base R. As we have mentioned in the introduction, base R string operations have traditionally been limited in scope. There also might be some issues with regards to their portability, reasons for which may be plentiful. For instance, varied variants of the **PCRE** (versions 8.x or 10.x thereof) pattern matching libraries may be linked to during the compilation of R. On Windows, there is a custom implementation of **iconv** that has a set of character encoding IDs not fully compatible with that on GNU/Linux: to select the Polish locale, we are required to pass "Polish_Poland" to `Sys.setlocale()` on Windows whereas "pl_PL" on Linux. Interestingly, R can be built against the system **ICU** so that it uses its Collator for comparing strings (e.g., using the "`<=`" operator), however this is only optional and does not provide access to any other Unicode services.

For example, let us consider the matching of "all letters" by means of the built-in `gregexpr()` function and the **TRE** (`perl=FALSE`) and **PCRE** (`perl=TRUE`) libraries using a POSIX-like and Unicode-style character set (see Section 5 for more details):

```
R> x <- "AEZaezĄĘŻąęż" # "AEZaez\u0104\u0118\u017b\u0105\u0119\u017c"
R> stri_sub(x, gregexpr("[[:alpha:]]", x, perl=FALSE)[[1]], length=1)
R> stri_sub(x, gregexpr("[[:alpha:]]", x, perl=TRUE)[[1]], length=1)
R> stri_sub(x, gregexpr("\\p{L}", x, perl=TRUE)[[1]], length=1)
```

On Ubuntu Linux 20.04 (UTF-8 locale), the respective outputs are:

```
[1] "A" "E" "Z" "a" "e" "z" "Ą" "Ę" "Ż" "ą" "ę" "ż"
[1] "A" "E" "Z" "a" "e" "z"
[1] "A" "E" "Z" "a" "e" "z" "Ą" "Ę" "Ż" "ą" "ę" "ż"
```

On Windows, when `x` is marked as UTF-8 (see Section 8.2), the current author obtained:

```
[1] "A" "E" "Z" "a" "e" "z"
[1] "A" "E" "Z" "a" "e" "z"
[1] "A" "E" "Z" "a" "e" "z" "Ą" "Ę" "Ż" "ą" "ę" "ż"
```

And again on Windows using the Polish locale but `x` marked as natively-encoded (CP-1250 in this case):

```
[1] "A" "E" "Z" "a" "e" "z" "Ę" "ę"
[1] "A" "E" "Z" "a" "e" "z" "Ą" "Ę" "Ż" "ą" "ę" "ż"
[1] "A" "E" "Z" "a" "e" "z" "Ę" "ę"
```

As we have mentioned in Section 6, when **stringi** links to **ICU** built from sources (`install.packages("stringi", configure.args="--disable-pkg-config")`), we are always guaranteed to get the same results on every platform.

High performance of stringi. Because of the aforementioned reasons, functions in **stringi** do not refer to their base R counterparts. A vast majority of them has been written in pure C and C++. What is more, the operations that do not rely on **ICU** services have been written from scratch with speed and portability in mind. For example, here are some timings of string concatenation:

```
R> x <- stri_rand_strings(length(LETTERS)*1000, 1000)
R> microbenchmark::microbenchmark(
+   join2=stri_join(LETTERS, x, sep="", collapse="", ),
+   join3=stri_join(x, LETTERS, x, sep="", collapse="", ),
+   r_paste2=paste(LETTERS, x, sep="", collapse="", ),
+   r_paste3=paste(x, LETTERS, x, sep="", collapse="", )
+ )
```

Unit: milliseconds

expr	min	lq	mean	median	uq	max	neval
join2	65.047	66.854	85.585	68.081	124.24	140.66	100
join3	150.471	152.858	161.106	154.994	173.48	227.91	100
r_paste2	189.242	193.616	219.677	200.867	257.52	306.17	100
r_paste3	398.432	409.254	470.062	496.744	509.85	526.54	100

Another example – timings of fixed pattern searching:

```
R> x <- stri_rand_strings(100, 100000, "[actg]")
R> y <- "acca"
R> microbenchmark::microbenchmark(
+   fixed=stri_locate_all_fixed(x, y),
+   regex=stri_locate_all_regex(x, y),
+   coll=stri_locate_all_coll(x, y),
+   r_tre=gregexpr(y, x),
+   r_pcre=gregexpr(y, x, perl=TRUE),
+   r_fixed=gregexpr(y, x, fixed=TRUE)
+ )
```

Unit: milliseconds

expr	min	lq	mean	median	uq	max	neval
fixed	8.5722	8.8406	9.0277	9.1003	9.1114	9.4158	100
regex	243.9435	251.2697	252.7098	254.0506	255.0880	264.6571	100
coll	934.9043	950.4080	973.5861	982.5842	990.0472	1013.1969	100
r_tre	285.8963	287.2784	293.8979	295.9722	296.6771	301.9577	100
r_pcre	194.7751	196.0178	200.3546	201.8720	202.6245	215.1482	100
r_fixed	109.4371	110.5815	113.2501	114.1326	114.3577	120.6593	100

stringi functions can also be accessed from within C++ code. Authors of statistical/data analysis software who would like to speed up their projects are kindly referred to the **ExampleR-cppStringi** package available at <https://github.com/gagolews/ExampleRcppStringi> for an illustration using **Rcpp** (Eddelbuettel 2013).

Future of stringi. Over the years, many useful R packages related to text processing have been developed, see (Feinerer, Hornik, and Meyer 2008; Welbers, Van Atteveldt, and Benoit 2017). Several of them are listed in the CRAN Task View on Natural Language Processing, see <https://cran.r-project.org/web/views/NaturalLanguageProcessing.html>. At the

time of writing of this paper, **stringi** itself has over 200 strong (direct) reverse dependencies and has established itself as one of the most frequently downloaded R extension whose user base is steadily growing.

Future work will involve the porting of **stringi** to different scientific/statistical computing environments, including Python with the **NumPy** (van der Walt, Colbert, and Varoquaux 2011) ecosystem, so as to provide more Unicode-aware alternatives to the vectorised text processing facilities from `numpy.char` and **pandas** (McKinney 2017, Chap. 7). Moreover, further extension of **stringi**'s API shall be conveyed so as to provide an even broader coverage of ICU services.

Acknowledgements

stringi is an open source project distributed under the terms of the BSD-3-clause license. Its most recent development snapshot is available through GitHub at <https://github.com/gagolews/stringi>. The bug- and feature request tracker can be accessed from <https://github.com/gagolews/stringi/issues>. Moreover, its homepage – which includes a reference manual and detailed documentation of the package's API – is located at <https://stringi.gagolewski.com/>.

The author wishes to thank Hadley Wickham for coming up with the original **stringr** package API (see Table 5). Also, greatly appreciated are the contributions of all who have donated their time and effort (in all the possible forms: code, feature suggestions, ideas, criticism, testing) to make **stringi** better – Bartek Tartanus, Kenneth Benoit, Marcin Bujarski, Bill Denney, Katrin Leinweber, Jeroen Ooms, Davis Vaughan, and many others, see <https://github.com/gagolews/stringi/graphs/contributors>. More contributions are always welcome.

References

- Chambers J (2008). *Software for Data Analysis. Programming with R*. Springer-Verlag.
- Clayden J (2019). **ore**: *An R Interface to the Onigmo Regular Expression Library*. R package version 1.6.3, URL <https://CRAN.R-project.org/package=ore>.
- Crochemore M, Rytter W (2003). *Jewels of Stringology. Text Algorithms*. World Scientific.
- Dasu T, Johnson T (2003). *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons.
- Davis M, Chapman C (2021). “Unicode Standard Annex #29: Unicode Text Segmentation.” URL <http://unicode.org/reports/tr29/>.
- Davis M, Heninger A (2021). “Unicode Technical Standard #18: Unicode Regular Expressions.” URL <http://www.unicode.org/reports/tr18/>.
- Davis M, Whistler K (2021). “Unicode Standard Annex #15: Unicode Normalization Forms.” URL <http://www.unicode.org/reports/tr15/>.

stringr	Base R	Purpose
<code>str_c()</code>	<code>paste()</code>	join strings
<code>str_count()</code>	<code>gregexpr()</code>	count pattern matches
<code>str_detect()</code>	<code>grepl()</code>	detect pattern matches
<code>str_dup()</code>		duplicate strings
<code>str_extract()</code> , <code>str_extract_all()</code>		extract (first, all) pattern matches
<code>str_length()</code>	<code>nchar()</code>	compute string length
<code>str_locate()</code> , <code>str_locate_all()</code>	<code>regexpr()</code> , <code>gregexpr()</code>	locate (first, all) pattern matches
<code>str_match()</code> , <code>str_match_all()</code>	<code>regexec()</code>	extract (first, all) matches to regex capture groups
<code>str_pad()</code>		add whitespaces at beginning or end
<code>str_trim()</code>	<code>trimws()</code>	remove whitespaces from beginning or end
<code>str_replace()</code> , <code>str_replace_all()</code>	<code>sub()</code> , <code>gsub()</code>	replace (first, all) pattern matches with a replacement string
<code>str_split()</code> , <code>str_split_fixed()</code>	<code>strsplit()</code>	split up a string into pieces
<code>str_sub()</code> , <code>`str_sub<-`()</code>	<code>substring()</code>	extract or replace substrings
<code>str_wrap()</code>	<code>strwrap()</code>	split strings into text lines
<code>word()</code>		extract words from a sentence

Table 5: Functions in (the historical) **stringr** 0.6.2 and their base R counterparts.

Davis M, Whistler K, Scherer M (2021). “Unicode Technical Standard #10: Unicode Collation Algorithm.” URL <http://www.unicode.org/reports/tr10/>.

Eddelbuettel D (2013). *Seamless R and C++ Integration with Rcpp*. Springer-Verlag, New York.

Feinerer I, Hornik K, Meyer D (2008). “Text Mining Infrastructure in R.” *Journal of Statistical Software*, **25**(5), 1–54.

Friedl JEF (2006). *Mastering Regular Expressions*. O’Reilly.

Hocking TD (2019). “Comparing **namedCapture** with Other R Packages for Regular Expressions.” *The R Journal*, **11**/2, 328–346.

Hopcroft JE, Ullman JD (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Jurafsky D, Martin JH (2008). *Speech and Language Processing*. Prentice Hall.

- Kleene SC (1951). “Representation of Events in Nerve Nets and Finite Automata.” *Technical Report RM-704*, The RAND Corporation, Santa Monica, CA. URL https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf.
- Knuth D, Morris JH, Pratt V (1977). “Fast Pattern Matching in Strings.” *SIAM Journal on Computing*, **6**(2), 323–350.
- Knuth D, Plass M (1981). “Breaking Paragraphs into Lines.” *Software: Practice and Experience*, **11**, 1119–1184.
- Kurtz S, *et al.* (2004). “Versatile and Open Software for Comparing Large Genomes.” *Genome Biology*, **5**, R12.
- McKinney W (2017). *Python for Data Analysis*. O’Reilly.
- Rabin M, Scott D (1959). “Finite Automata and Their Decision Problems.” *IBM Journal of Research and Development*, **3**, 114–125.
- R Development Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. <http://www.R-project.org>.
- Ritchie DM, Thompson KL (1970). “**QED** Text Editor.” *Technical Report 70107-002*, Bell Telephone Laboratories, Inc. URL <https://wayback.archive-it.org/all/20150203071645/http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf>.
- Szpankowski W (2001). *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons.
- van der Loo M (2014). “The **stringdist** Package for Approximate String Matching.” *The R Journal*, **6**(1), 111–122.
- van der Loo M, de Jonge E (2018). *Statistical Data Cleaning with Applications in R*. John Wiley & Sons.
- van der Walt S, Colbert SC, Varoquaux G (2011). “The **NumPy** Array: A Structure for Efficient Numerical Computation.” *Computing in Science Engineering*, **13**(2), 22–30.
- Welbers K, Van Attevelde W, Benoit K (2017). “Text Analysis in R.” *Communication Methods and Measures*, **11**(4), 245–265.
- Wenfeng Q (2020). **re2r: RE2 Regular Expression**. R package version 1.0.0, URL <https://github.com/qinwf/re2r>.
- Whistler K, Iancu L (2021). “Unicode Standard Annex #44: Unicode Character Database.” URL <http://unicode.org/reports/tr44/>.
- Wickham H (2010). “**stringr**: Modern, Consistent String Processing.” *The R Journal*, **2**(2), 38–40.
- Wickham H (2020). **rvest: Easily Harvest (Scrape) Web Pages**. R package version 0.3.6, URL <https://CRAN.R-project.org/package=rvest>.
- Wickham H, Grolemund G (2017). *R for Data Science*. O’Reilly.

Wickham H, Hester J, Ooms J (2020). **xml2**: *Parse XML*. R package version 1.3.2, URL <https://CRAN.R-project.org/package=xml2>.

Xie Y (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC.

Affiliation:

Marek Gagolewski

School of Information Technology

Deakin University

Geelong, VIC 3220, Australia

E-mail: m.gagolewski@deakin.edu.au

URL: <https://www.gagolewski.com/>