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, should be included in each statistician's or data scientist's repertoire.

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 September 7, 2020.

Please cite as: Gagolewski M (2020). ***stringi**: Fast and Portable Character String Processing in R*. URL `https://stringi.gagolewski.com`.

1. Introduction

Stringology, see (Crochemore and Rytter 2003), deals with algorithms and data structures used for processing of character strings (Jurafsky and Martin 2008; Szpankowski 2001). From the perspective of applied statistics and data science, it is worth stressing that many interesting datasets first come in an unstructured or contaminated form, for instance when fetched from different APIs or when gathered by means of web scraping techniques. Diverse data cleansing and preparation operations (Dasu and Johnson 2003; van der Loo and de Jonge 2018) 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. Amongst them we may find: string concatenation, substring extraction, collation, sorting, Unicode normalisation, transliteration, pattern matching, and date-time parsing. Activities related to information retrieval, computer vision, bioinformatics, natural language processing, or even musicology can also benefit from including them in the data processing pipelines, see (Jurafsky and Martin 2008; Kurtz *et al.* 2004).

Base R (R Development Core Team 2020) provides a few functions for dealing with character strings, see (Chambers 2008, Chapter 8) and Table 1. However, it is the **stringr** package (Wickham 2010), first released in November 2009, that marks the first milestone of implementing the idea of a "tidier" API for text data processing. In version 0.6.2 (dated 2012–12–06) of **stringr**'s `README`, we read that this package:

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

Table 1: Functions in (the historical) **stringr** 0.6.2 and their base R counterparts.

- *processes factors and characters in the same way,*
- *gives functions consistent names and arguments,*
- *simplifies string operations by eliminating options that you don't need 95% of the time,*
- *produces outputs than can easily be used as inputs. This includes ensuring that missing inputs result in missing outputs, and zero length inputs result in zero length outputs,*
- *completes R's string handling functions with useful functions from other programming languages.*

The list of the functions available in **stringr** at that time is given in Table 1.

Nevertheless, **stringr** was developed as a set of wrappers around its base R counterparts, which not only limited its scope but also could cause many portability issues. In particular, the same code may yield different results on different operating systems, some services such as the processing of particular languages may be unavailable whatsoever, etc. For instance,

varied versions of the **PCRE** (versions 8.x or 10.x thereof) pattern matching libraries may be linked to during compilation. 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. Moreover, R can be build against the system **ICU** so that it uses its Collator for comparing strings, however this is only optional.

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 4 for more details):

```
R> library("stringi")  # substring extraction with stri_sub(), see below
R> x <- "AEZaezĄĘŻąęż"
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, we get:

```
[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"
```

In order to overcome such portability problems, in 2013 I have developed the **stringi** package (pronounced "stringy", IPA [stringi]). Its API has been designed so as to be compatible and consistent with that of **stringr**'s, which has already proven effective and convenient. All the functions have been written from scratch to guarantee that they are as efficient as possible. For the processing of text in different languages, which are plentiful, the **ICU** library (see http://site.icu-project.org/) is relied upon to assure full conformance to the Unicode standards.

Over the years, **stringi** confirmed itself as robust, production quality software. Interestingly, from version 1.0, **stringr** has been rewritten as a set of wrappers around **stringi** instead of base R routines; it aims to be more beginner-friendly, see (Wickham and Grolemund 2017, Chapter 14). On the other hand, **stringi** provides many more functions (250 vs. 52); some of

them are more specialised or equipped with more control parameters to enable fine-tuning. This paper describes the facilities provided by **stringi** in-depth so that the package's users can get the most out of them. It also demonstrates the wide range of tools that more advanced statisticians and data scientists may find useful in their daily activities.

What remains is set out as follows. Basic string operations such as substring extraction and concatenation are discussed in Section 2. Section 3 discusses searching for fixed substrings. Section 4 details pattern matching with **ICU** regular expressions. Section 5 deals with the locale-aware **ICU** Collator, suitable for natural language processing activities. Section 6 introduces other operations such as text boundary analysis or date-time formatting and parsing. Section 7 details on encoding conversion and detection as well as Unicode normalisation. Finally, Section 8 concludes the paper.

All the code chunks' outputs presented in this paper were obtained using R 4.0.2. The R environment itself and all the packages used herein are available from CRAN at https://CRAN.R-project.org/.

```
R> # install.packages("stringi")  # to download from CRAN and install
R> library("stringi")  # load and attach the package's namespace
```

Here we describe **stringi** 1.5.2, which has been built against the following version of the **ICU** library:

```
R> cat(stri_info(short=TRUE))

stringi_1.5.2 (en_AU.UTF-8; ICU4C 61.1 [bundle]; Unicode 10.0)
```

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 detailed documentation of the package's API – is located at https://stringi.gagolewski.com/.

2. Basic string operations

In order to get started with string processing in the R environment, we should first emphasise that the language itself does not provide access to 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 quite convenient from the practical side; the so-called *vectorisation* strategy encourages writing of code for processing *whole* collections of objects all at once, regardless of their size. For example, let's consider the following data frame:

```
R> (birth_dates <- head(
+      # see https://github.com/gagolews/stringi/tree/master/datasets
+      read.csv(header=TRUE, comment="#", file="birth_dates.csv"), 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.

2.1. Computing length and width

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

Length. Let's 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 7.1 for more details.

```
R> stri_length(x)

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

`stri_length(x)` returns a numeric vector `l`, with the same number of elements as `x`, such that, for every `i`, `l[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, so 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 an automatically generated report), a string's *width* estimate – an approximate number of text columns it occupies when printed using a mono-spaced font – may be more informative. In particular, many CJK (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
```

2.2. Joining

Below we describe the functions that are based on string concatenation.

Operator %s+%. To join (concatenate) 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 and 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 in the case of the built-in `paste()` function):

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

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

For dealing with missing values, we may use 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, "!")  # sep="", collapse=NULL

[1] "X1!" "Y2!" "Z3!" "X4!" "Y5!" "Z6!"

R> stri_join(c("X", "Y", "Z"), 1:6, "+", sep=".", collapse="; ")

[1] "X.1.+; Y.2.+; Z.3.+; X.4.+; Y.5.+; Z.6.+"
```

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, 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"         "spamspam"     "spamspamspam"

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.

2.3. Extracting and replacing substrings

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

Indexing vectors. In order to pick a subsequence from any R vector, we use the square-bracket operator[1] with an index vector consisting of either non-negative integers, negative integers, or logical values[2].

```
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"
```

[1] More 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 `sapply(x, "[[", 1)`.

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

`stri_sub_all()`. The `stri_sub()` function is of course vectorised with respect to all its arguments (the 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's 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's 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 with 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"
```

3. 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 Unicode[3] 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.

3.1. 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, etc.

[1] FALSE  TRUE FALSE    NA
```

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

3.2. Searching for fixed strings

Table 2 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 4 and Section 5).

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).

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

```
R> stri_count_fixed("abcabcdefabcabcabdc", "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("abcabcdefabcabcabdc", fixed="abc")

[1] 4
```

[3] All functions in **stringi** automatically convert all R strings to Unicode, see Section 7.2 for discussion.

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

Table 2: 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 3), `stri_detect_regex()` (see Section 4), and `stri_split_coll()` (see Section 5).

Vectorisation. 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. The users, unaware of this rule, might find this behaviour unintuitive at the beginning, especially if something does not go the way they expect. Therefore, let us point out 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
```

Search engine options. The pattern matching engine may be tuned by passing further arguments to the search functions (via "`...`"; they are be redirected as-is to `stri_opts_fixed()`). Table 3 gives the list of available options.

First, we may turn on the simplistic[4] 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
```

[4] Which 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 5 (and partially Section 4) for a workaround.

Option	Purpose
`case_insensitive`	logical; whether to enable the simple case-insensitive matching (defaults to `FALSE`)
`overlap`	logical; whether to enable the detection of overlapping matches (defaults to `FALSE`)

Table 3: Options for the fixed pattern search engine, see `stri_opts_fixed()`.

Detecting and subsetting patterns. A somehow simplified version of the above 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):

```
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 ends[5] with a pattern match:

```
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
```

[5]Note 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.

```
[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")]`. Moreover:

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

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

```
R> stri_subset_fixed(x, "abc", negate=TRUE)  # all 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      ""
```

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, "agggAGAGAgaGAtcga", "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,]     5   7
[2,]     7   9
[3,]     9  11
[4,]    11  13

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

Let us recall that such kinds of "from-to" matrices 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
```

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, "agggAGAGAgaGAtcga", "agagagaga")
R> stri_replace_first_fixed(x, "aga", "~", case_insensitive=TRUE)

[1] "~"               "actg"            NA                "aggg~GAgaGAtcga"
[5] "~gagaga"

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

[1] "~"               "actg"            NA                "agggAGAGAg~tcga"
[5] "agagag~"

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

[1] "~"             "actg"          NA              "aggg~G~GAtcga"
[5] "~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 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.

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"
```

4. Regular expressions

Regular expressions (*regex*es) 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. standalone 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 `regexec()`, see Table 1):

- ERE[6] (*extended regular expressions* that conform to the **POSIX.2-1992** standard); used by default,
- PCRE[7] (*Perl-compatible regular expressions*), in use if `perl = TRUE` is set.

Other matchers are implemented in the **ore** (via the **Onigmo** library) and **re2r** (**RE2**) packages.

Stringi, on the other hand, provides access to the regex engine implemented in **ICU**, which was inspired by `java.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 patters). On the other hand, **ICU** regexes fully conform to the Unicode Technical Standard #18 (Davis and Heninger 2020) and hence provide comprehensive support for Unicode.

It is worth noting that most programming languages as well as advanced text editors and IDEs (including **RStudio**) allow for finding or replacing patters 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 with the `*_regex()` suffix. Apart from `stri_detect_regex()`, `stri_locate_all_regex()`, and so forth, in Section 4.4 we introduce `stri_match_all_regex()`, `stri_match_first_regex()`, and `stri_match_last_regex()`. Moreover, Table 4 lists the available options for the regex engine.

4.1. 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.

First, let's note that 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
```

[6] Via the **TRE** library (`https://github.com/laurikari/tre/`).
[7] Via the **PCRE2** library (`https://www.pcre.org/`).

Option	Purpose
`case_insensitive` [regex flag `(?i)`]	logical; defaults to `FALSE`; whether to enable (full) case-insensitive matching
`comments` [regex flag `(?x)`]	logical; defaults to `FALSE`; whether to allow white spaces and comments within patterns
`dot_all` [regex flag `(?s)`]	logical; defaults to `FALSE`; if set, "`.`" matches line terminators, otherwise its matching stops at a line end
`literal`	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 (`*_fixed()` functions described in Section 3) are faster
`multi_line` [regex flag `(?m)`]	logical; defaults to `FALSE`; if set, "`$`" and "`^`" recognise line terminators within a string, otherwise, they match only at start and end of the input
`unix_lines`	logical; defaults to `FALSE`; when enabled, only the Unix line ending, i.e., `U+000a`, is honoured as a terminator by "`.`", "`$`", and "`^`"
`uword` [regex flag `(?w)`]	logical; defaults to `FALSE`; whether to use the Unicode definition of word boundaries (see Section 6.1), which are quite different from the traditional regex word boundaries
`error_on_unknown_escapes`	logical; defaults to `FALSE`; whether unrecognised backslash-escaped characters trigger an error; by default, unknown backslash-escaped ASCII letters represent themselves
`time_limit`	integer; processing time limit for match operations in ∼milliseconds (depends on the CPU speed); 0 for no limit (the default)
`stack_limit`	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 4: Options for the regular expressions search engine, see `stri_opts_regex()`.

However, this time the case insensitive mode fully supports Unicode matching[8]:

```
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

[8]This does not mean, though, that it considers canonically equivalent strings as equal, see Section 5.2 for discussion and a workaround.

literally – we have to escape it with a backslash, "\". Yet, the backlash 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)

[[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, "[hj]am")

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

the "`[hj]am`" 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 7.1 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"
```

Using predefined character sets. Each code point is assigned a unique *general category*, which can be thought of a character's class, see (Whistler and Iancu 2020). Sets of characters from each category can be referred to, amongst others, by using the "`\p{category}`" syntax:

```
R> x <- "aąbßÆAĄB 你 123,.;'! \t-+=[]©←→" „²³¾"
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

[[1]]
[1] "Æ" "A" "Ą" "B"

R> stri_extract_all_regex(x, "\\p{N}")   # number

[[1]]
[1] "1" "2" "3" "²" "³" "¾"

R> stri_extract_all_regex(x, "\\p{P}")   # punctuation

[[1]]
 [1] "," "." ";" "'" "!" "-" "[" "]" "”" " „"

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

[[1]]
[1] "+" "=" "©" "←" "→"
```

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

```
R> stri_extract_all_regex(x, "\\w")     # word characters
```

```
[[1]]
 [1] "a"  "ą"  "b"  "ß"  "Æ"  "A"  "Ą"  "B"  " 你" "1"  "2"  "3"

R> stri_extract_all_regex(x, "\\d")     # decimal digits, \p{Nd}

[[1]]
[1] "1" "2" "3"

R> stri_extract_all_regex(x, "\\s")     # spaces, [\t\n\f\r\p{Z}]

[[1]]
[1] " "  "\t"
```

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

Avoiding POSIX classes. The use of the POSIX-like character classes should be avoided. The **ICU** User Guide states that in general they are not well-defined.

In particular, in POSIX-like regex engines, "`[:punct:]`" stands for the character class corresponding to the `ispunct()` function in `C` (check out `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 <- ",./|\\<>?;:'\"[]{}-=_+()*&^%$€#@!`~×„” "
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] "€" "×" " " "„" "” "
```

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] "|" "<" ">" "=" "+" "^" "$" "€" "`" "~" "×"

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

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

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

4.2. 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"
```

It has a very low precedence. Therefore, if we wish to introduce an alternative of *sub*expressions, we need to group them, e.g., between round brackets[9]:

```
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"
```

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

Non-grouping parentheses. Some parenthesised subexpressions where the opening bracket is followed by the question mark have 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"
```

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 refer to Table 4.

4.3. Quantifiers

Oftentimes, we need to enable the matching of a variable number of instances of the same subexpression or make its presence totally 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+`" is matched by `"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
 22.257   0.000  22.263
```

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** package (a wrapper around the **RE2** library) documentation and the references therein for discussion.

4.4. 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. This is most evident when we use the capture group-sensitive versions of `stri_extract()`: `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'" "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.

Although, in general, parsing of 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 discussion.

4.5. Anchoring

Lastly, let us discuss 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 numbers[10]:

```
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"
```

[10] This regex is for didactic purposes only.

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"
```

5. 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 point vs. code point matching does *not* take into account accented and conjoined letters as well as ignorable punctuation and case.

The **ICU** Collation Service[11] provides the basis for such string comparison activities 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 2020), we may expect that the generated results will meet the requirements of the culturally correct natural language processing in any *locale*.

5.1. 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 specifier[12] 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-630-2 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

[11] See the **ICU** User Guide on Collation, http://userguide.icu-project.org/collation.

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

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 6.5 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] 722
```

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

```
[1] "pt_PT"  "seh_MZ" "fr_GP"  "en_RW"  "en_IO"
```

Querying for locale-specific services. The availability of locale-specific services can only be determined during the very request for a particular resource[13]. 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 eagerly[14]. 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.

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

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

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 5) 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"
```

5.2. 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 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
```

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

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

[1] "Gągolewski" "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
```

5.3. Linear ordering of strings

Operators such that `%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")
```

Option	Purpose
`locale`	a string specifying the locale to use; `NULL` (default) or `""` for the current default locale as indicated by `stri_locale_get()`
`strength`	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
`uppercase_first`	logical; `NA` (default) orders upper and lower case letters in accordance to their tertiary weights, `TRUE` forces upper case letters to sort before lower case letters, `FALSE` does the opposite
`numeric`	logical; if `TRUE`, a collation key for the numeric value of substrings of digits is generated; this is a way to make `"100"` ordered after `"2"`; defaults to `FALSE`
`case_level`	logical; if `TRUE`, an extra case level (positioned before the third level) is generated; defaults to `FALSE`
`normalisation`	logical; if `TRUE`, then an incremental check is performed to see whether the input data is in the FCD ("fast C or D") form; if the data is not in the FCD form, the incremental NFD normalisation is performed, see Section 7.4; defaults to `FALSE`
`alternate_shifted`	logical; if `FALSE` (default), all code points with non-ignorable primary weights are handled in the same way; `TRUE` causes the code points with primary weights that are equal or below 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 examples provided
`french`	logical; `TRUE` results in secondary weights being considered backwards, i.e., ordering according to the last accent difference – nowadays only used in Canadian French; defaults to `FALSE`

Table 5: Options for the **ICU** Collator that can be passed to `stri_opts_collator()`.

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

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

5.4. Collator options

Table 5 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.* 2020) can go beyond simple canonical equivalence and allow us to treat some other (depending on the context) differences as *negligible*.

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
```

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 different 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)                  # ß == ss, case insensitive

[1] "groß"

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

[1] "groß"  "GROSS" "Groß"

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

[1] "groß"  "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", "côte", "coté", "côté"), locale="fr_FR")

[1] "cote" "coté" "côte" "côté"

R> stri_sort(c("cote", "côte", "coté", "côté"), locale="fr_CA") # french=TRUE

[1] "cote" "côte" "coté" "côté"
```

Sorting numerals. Moreover, let's 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 7.4 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 2020) also discusses the so-called *compatibility* equivalence of strings. This is a looser form of similarity; it is observed when 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 7.4 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.

5.5. String searching

The **ICU** Collator can also be utilised when there is a need to locate the occurrences of simple textual patterns. All the string search functions described in Section 3 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
```

6. Other operations

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

6.1. 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 2020). 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 very useful. " %s+%
+       "My hovercraft is full of eels, spam, eggs, and spam."
R> stri_count_boundaries(x, type="sentence")  # number of sentences
```

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

```
[1] 2

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

[1] 41

R> stri_count_words(x)                    # number of words themselves

[1] 17

R> stri_extract_all_words(x)

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

6.2. Trimming, padding, and other formatting

The following functions can be useful when pretty-printing character strings or text on the console, dynamically generating reports (compare **knitr**'s `results = "asis"` chunk option), or creating text files (e.g., with `stri_write_lines()`, see Section 7.3).

Padding. Strings can be padded with some character so that they are of the desired lengths by means of the `stri_pad()` function. 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 6.1).

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'"
```

6.3. 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] " ヲグムノタルソタ"
```

See Section 4.1 for different ways to specify character sets.

6.4. 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("Iİ", locale = "tr_TR")               # Turkish

[1] "ıi"

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-Telugu"     "Devanagari-Arabic"  "Malayalam-Tamil"
[4] "Any-uz/BGN"         "Any-Greek"          "dv-dv_Latn/BGN"
[7] "Malayalam-Gurmukhi" "Gujr-Latn"          "Gujarati-Kannada"
```

Some examples:

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

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

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

[1] "Let’ s go…— that’ s what she said."
```

6.5. Parsing and formatting date and time

In base R, dealing with dates and times in languages different than the current locale is somewhat difficult. For instance, most of the readers of this paper may 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 parsing/formatting temporal data from/to any locale is possible:

```
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 patterns[17]. For convenience, `strftime()`- and `strptime()`-compatible 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")
```

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

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

```
[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] "Freitag, 4. September 2020 um 07:11:49 Mitteleuropäische Sommerzeit"

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 5:11:49 PM NZST"

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] " 平成 32 年 2 月 4 日火曜日" " 平成 32 年 8 月 7 日金曜日"
```

7. 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.

7.1. 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 4.1), such as letters, numbers, etc.

```
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
 125093    2063    1702    1502     770    6978  128238     590      25
```

Yet, most of the code points are still unallocated – the Unicode standard is updated from time to time, e.g., the 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 $\leq 127 = 7\mathrm{F}_{16}$). 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; $7\mathrm{A}_{16}$ 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 in with another backslash).

```
R> x <- "zß 你好"
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 XeLaTeX 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}"
```

7.2. 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 <- "abz0ąß 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** always[18] outputs UTF-8 data.

7.3. Reading and writing text files and converting between encodings

According to a report by W3Techs[19], as of 2020–09-04, 95.4% of websites use UTF-8. Nevertheless, encountering other encodings is still quite likely.

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

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

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> # see https://github.com/gagolews/stringi/tree/master/datasets
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.

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 build-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"`.

7.4. Normalising strings

In Section 5.2 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 bytewise matching (Davis and Whistler 2020).

```
R> x <- "a\u0328 ą" # 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 expected:

```
R> stri_trans_nfkd("r²⁀")

[1] "r2{"
```

8. Conclusion

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). Many 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.

The complete documentation of the package's API is available at https://stringi.gagolewski.com/. **stringi** functions can also be accessed from within C++ code. See the **ExampleRcppStringi** package available at https://github.com/gagolews/ExampleRcppStringi for an example using **Rcpp** (Eddelbuettel 2013).

Finally, it is worth stressing that functions in **stringi** are not wrappers around base R facilities. A vast majority of them has been written in pure C and C++. 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  40.822  41.308  54.693  42.604  81.871  94.722   100
    join3  81.918  94.066  98.401  95.388 103.573 145.161   100
 r_paste2 108.930 111.723 129.644 114.045 161.575 189.563   100
 r_paste3 228.921 235.284 276.171 293.859 301.363 321.197   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   5.2353   5.3325   5.4448   5.3723   5.4856   6.8424   100
   regex 121.0332 122.0588 122.9537 122.5455 123.4710 129.5061   100
    coll 395.9836 397.9740 399.4649 398.9458 400.1369 414.9644   100
   r_tre 134.5434 135.7514 136.7537 136.2900 137.2321 144.3329   100
  r_pcre  83.3282  84.4817  85.2377  85.1256  85.6796  88.2810   100
 r_fixed  55.8149  56.2731  56.7070  56.5676  57.0615  61.2604   100
```

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

Acknowledgements

First and foremost, the author wishes to thank Hadley Wickham for coming up with the **stringr** package API and bringing the idea of "tidying up" R string processing workflows. Also, many thanks to all the contributors who have donated their time and effort (in all the possible forms: code, feature suggestions, ideas, criticism) 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.

Crochemore M, Rytter W (2003). *Jewels of Stringology. Text Algorithms.* World Scientific.

Dasu T, Johnson T (2003). *Exploratory Data Mining and Data Cleaning.* Wiley & Sons.

Davis M, Chapman C (2020). "Unicode Standard Annex #29: Unicode Text Segmentation." URL http://unicode.org/reports/tr29/.

Davis M, Heninger A (2020). "Unicode Technical Standard #18: Unicode Regular Expressions." URL http://www.unicode.org/reports/tr18/.

Davis M, Whistler K (2020). "Unicode Standard Annex #15: Unicode Normalization Forms." URL http://www.unicode.org/reports/tr15/.

Davis M, Whistler K, Scherer M (2020). "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 (2020). *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*. 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*. Wiley & Sons.

Welbers K, Van Atteveldt W, Benoit K (2017). "Text Analysis in R." *Communication Methods and Measures*, **11**(4), 245–265.

Whistler K, Iancu L (2020). "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, Grolemund G (2017). *R for Data Science*. O'Reilly.

Affiliation:

Marek Gagolewski
School of Information Technnology
Deakin University
Geelong, VIC 3220, Australia
E-mail: m.gagolewski@deakin.edu.au
URL: https://www.gagolewski.com/