stringi: Fast and Portable Character String Processing in R

Marek Gagolewski
Deakin University, Geelong, VIC 3220, Australia
Systems Research Institute, Polish Academy of Sciences

Abstract

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

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

This is a draft version of a paper on **stringi**, last updated on July 20, 2021.

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

1. Introduction

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

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

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

Most statistical computing ecosystems provide only a basic set of text operations. In particu-

lar, base R (R Development Core Team 2021) is mostly restricted to pattern matching, string concatenation, substring extraction, trimming, padding, wrapping, simple character case conversion, and string collation, see (Chambers 2008, Chapter 8) and Table 5 below. The **stringr** package (Wickham 2010), first released in November 2009, implemented an alternative, "tidy" API for text data processing (cleaned-up function names, more beginner-friendly outputs, etc.; the list of 21 functions that were available in **stringr** at that time is given in Table 5). The early **stringr** featured a few wrappers around a subset of its base R counterparts. The latter, however – to this day – not only is of limited scope, but also suffers from a number of portability issues; it may happen that the same code can yield different results on different operating systems; see Section 3 for some examples.

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

Over the years, **stringi** confirmed itself as robust, production-quality software; for many years now it has been one of the most often downloaded R extensions. Interestingly, in 2015 the aforementioned **stringr** package has been rewritten as a set of wrappers around some of the **stringi** functions instead of the base R ones. In Section 14.7 of *R for Data Science* (Wickham and Grolemund 2017) we read: ***stringr*** *is useful when you're learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions.* ***stringi****, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need:* ***stringi*** *has 250 functions to* ***stringr****'s 49.* Also, it is worth noting that the recently-introduced **stringx** package (Gagolewski 2021) supplies a **stringi**-based set of portable and efficient replacements for and enhancements of the base `R` functions.

This paper describes the most noteworthy facilities provided by **stringi** that statisticians and data analysts may find useful in their daily activities. We demonstrate how important it is for a modern data scientist to be aware of the challenges of natural language processing in the internet era: how to force `"groß"` compare equal to `"GROSS"`, count the number of occurrences of `"AGA"` within `"ACTGAGAGACGGGTTAGAGACT"`, make `"a13"` ordered before `"a100"`, or convert between `"GRINNING FACE"` and `"😀"` forth and back. Such operations are performed by the very **ICU** itself; we therefore believe that what follows may be of interest to data-oriented practitioners employing Python, Perl, Julia, PHP, etc., as **ICU** has bindings for many other languages.

Here is the outline of the paper:

- Section 2 illustrates the importance of string processing in an example data preparation activity.
- General package design principles are outlined in Section 3, including the use cases of deep vectorisation, the concepts of data flow, and the main deviations from base R (also

with regards to portability and speed).

- Basic string operations, such as computing length and width of strings, string concatenation, extracting and replacing substrings, are discussed in Section 4.

- Section 5 discusses searching for fixed substrings: counting the number of matches, locating their positions, replacing them with other data, and splitting strings into tokens.

- Section 6 details **ICU** regular expressions, which are a powerful tool for matching patterns defined in a more abstract way, e.g., extracting numbers from text so that they can be processed quantitatively, identifying hyperlinks, etc. We show where **ICU** is different from other libraries like **PCRE**; in particular that it enables portable, Unicode-correct look-ups, for instance, involving sequences of emojis or mathematical symbols.

- Section 7 deals with the locale-aware **ICU** Collator, which is suitable for natural language processing activities; this is where we demonstrate that text processing in different languages or regions is governed by quite diverse rules, deviating significantly from the US-ASCII ("C/POSIX.1") setting. The operations discussed therein include testing string equivalence (which can turn out useful when we scrape data that consist of non-normalised strings, ignorable punctuation, or accented characters) as well as arranging strings with regards to different linear orderings.

- Section 8 covers some other useful operations such as text boundary analysis (for splitting text into words or sentences), trimming, padding, and other formatting, random string generation, character transliteration (converting between cases and alphabets, removing diacritic marks, etc.) as well as date-time formatting and parsing in any locale (e.g., Japanese dates in a German R).

- Section 9 details on encoding conversion and detection (which is key when reading or writing text files that are to be communicated across different systems) as well as Unicode normalisation (which can be useful for removing formatting distinctions from text, e.g., superscripts or font variants).

- Finally, Section 10 concludes the paper.

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

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

All code chunk outputs presented in this paper were obtained in R 4.1.0. The R environment itself and all the packages used herein are available from CRAN at https://CRAN.R-project.org/. In particular, `install.packages("stringi")` can be called to fetch the object of our focus. By calling:

```
R> library("stringi")
R> cat(stri_info(short=TRUE))

stringi_1.7.3 (en_AU.UTF-8; ICU4C 69.1 [bundle]; Unicode 13.0)
```

we can load and attach the package's namespace and display some basic information thereon. Hence, below we shall be working with **stringi** 1.7.3, however, as the package API is considered stable, the presented material should be relevant to any later versions.

2. Use case: Data preparation

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

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

The parsing of the city's *Wikipedia* page can be done by means of the functions from the **xml2** (Wickham, Hester, and Ooms 2020) and **rvest** (Wickham 2021) packages.

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

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

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

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

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

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

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

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

```
[1] 4
```

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

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

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

```
5                                                 11.6(52.9)
6                                                 -2.8(27.0)
7                                               600.9(23.66)
8                                                      90.6
9                                                        51
10                                                    2,191
11              Source: Bureau of Meteorology.[85][86][87]
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or the average daily precipitation:

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

```
[1] 1.6463
```

and so forth.

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

3. General design principles

The API of the early releases of **stringi** has been designed so as to be fairly compatible with that of the 0.6.2 version of the **stringr** package (Wickham 2010) (dated 2012; see Table 5), with some fixes in the consistency of the handling of missing values and zero-length vectors,

amongst others. However, instead of being merely thin wrappers around base R functions, which we have identified as not necessarily portable across platforms and not really suitable for natural language processing tasks, all the functionality has been implemented from the ground up, with the use of **ICU** services wherever applicable. Since the initial release, an abundance of new features has been added and the package can now be considered a comprehensive workhorse for text data processing. Note that the **stringi** API is stable. Future releases are aiming for as much backward compatibility as possible so that other software projects can safely rely on it.

3.1. Naming

Function and argument names use a combination of lowercase letters and underscores (and no dots). To avoid namespace clashes, all function names feature the "`stri_`" prefix. Names are fairly self-explanatory, e.g., `stri_locate_first_regex` and `stri_locate_all_fixed` find, respectively, the first match to a regular expression and all occurrences of a substring as-is.

3.2. Vectorisation

Individual character (or code point) strings can be entered using double quotes or apostrophes:

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

[1] "spam"
```

However, as the R language does not feature any classical scalar types, strings are wrapped around atomic vectors of type "`character`":

```
R> typeof("spam")  # object type; see also is.character() and is.vector()

[1] "character"

R> length("spam")  # how many strings are in this vector?

[1] 1
```

Hence, we will be using the terms "string" and "character vector of length 1" interchangeably.

Not having a separate scalar type is very convenient; the so-called *vectorisation* strategy encourages writing code that processes *whole* collections of objects, all at once, regardless of their size.

For instance, given the following character vector:

```
R> pythons <- c("Graham Chapman", "John Cleese", "Terry Gilliam",
+    "Eric Idle", "Terry Jones", "Michael Palin")
```

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> (pythons <- stri_split_fixed(pythons, " ", simplify=TRUE))
```

```
     [,1]      [,2]
[1,] "Graham"  "Chapman"
[2,] "John"    "Cleese"
[3,] "Terry"   "Gilliam"
[4,] "Eric"    "Idle"
[5,] "Terry"   "Jones"
[6,] "Michael" "Palin"
```

Due to vectorisation, we can generally avoid using the `for`- and `while`-loops ("for each string in a vector…"), which makes the code much more readable, maintainable, and faster to execute.

3.3. Acting Elementwise with Recycling

Binary and higher-arity operations in R are oftentimes vectorised with respect to all arguments (or at least to the crucial, non-optional ones). As a prototype, let us consider the binary arithmetic, logical, or comparison operators (and, to some extent, `paste()`, `strrep()`, and more generally `mapply()`), for example the multiplication:

```
R> c(10, -1) * c(1, 2, 3, 4)  # == c(10, -1, 10, -1) * c(1, 2, 3, 4)
```

```
[1] 10 -2 30 -4
```

Calling "`x * y`" multiplies the *corresponding* components of the two vectors elementwisely. As one operand happens to be shorter than another, the former is recycled as many times as necessary to match the length of the latter (there would be a warning if partial recycling occurred). Also, acting on a zero-length input always yields an empty vector.

All functions in **stringi** follow this convention (with some obvious exceptions, such as the `collapse` argument in `stri_join()`, `locale` in `stri_datetime_parse()`, etc.). In particular, all string search functions are vectorised with respect to both the *haystack* and the *needle* arguments (and, e.g., the *replacement* string, if applicable).

Some users, unaware of this rule, might find this behaviour unintuitive at the beginning and thus miss out on how powerful it is. Therefore, let us enumerate the most noteworthy scenarios that are possible thanks to the arguments' recycling, using the call to `stri_count_fixed(haystack, needle)` (which looks for a needle in a haystack) as an illustration:

- many strings – one pattern:

  ```
  R> stri_count_fixed(c("abcd", "abcabc", "abdc", "dab", NA), "abc")
  ```

  ```
  [1]  1  2  0  0 NA
  ```

  (there is 1 occurrence of `"abc"` in `"abcd"`, 2 in `"abcabc"`, and so forth)

- one string – many patterns:

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

[1]  0  2  2 NA
```

("def" does not occur in "abcdeabc", "bc" can be found therein twice, etc.)

- each string – its own corresponding pattern:

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

[1] 2 0 1
```

(there are two "a"s in "abca", no "z" in "def", and one "h" in "ghi")

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

```
R> (haystack <- matrix(  # example input
+    do.call(stri_join,
+      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> needle <- c("a", "b", "c")
R> matrix(stri_count_fixed(haystack, needle),  # call to stringi
+    nrow=3, dimnames=list(needle, NULL))

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

(this looks for "a" in the 1st row of haystack, "b" in the 2nd row, and "c" in the 3rd; in particular, there are 3 "a"s in "aaa", 2 in "aba", and 1 "b" in "baa"; this is possible due to the fact that matrices are represented as "flat" vectors of length nrow*ncol, whose elements are read in a column-major (Fortran) order; therefore, here, pattern "a" is being sought in the 1st, 4th, 7th, … string in haystack, 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(t(haystack), needle))).

- all strings – all patterns:

```
R> haystack <- c("aaa", "bbb", "ccc", "abc", "cba", "aab", "bab", "acc")
R> needle <- c("a", "b", "c")
R> structure(
+    outer(haystack, needle, stri_count_fixed),
+    dimnames=list(haystack, needle))  # 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
```

(which computes the counts over the Cartesian product of the two arguments)

3.4. Missing values

Some base `R` string processing functions, e.g., `paste()`, treat missing values as literal `"NA"` strings. **stringi**, however, *does* enforce the consistent propagation of missing values (like arithmetic operations):

```
R> paste(c(NA_character_, "b", "c"), "x", 1:2)  # base R

[1] "NA x 1" "b x 2"  "c x 1"

R> stri_join(c(NA_character_, "b", "c"), "x", 1:2)  # stringi

Warning: longer object length is not a multiple of shorter object length

[1] NA    "bx2" "cx1"
```

For dealing with missing values, we may rely on the convenience functions such as `stri_omit_na()` or `stri_replace_na()`.

3.5. Data flow

All vector-like arguments (including factors and objects) in **stringi** are treated in the same manner: for example, if a function expects a character vector on input and an object of other type is provided, `as.character()` is called first (we see that in the example above, "`1:2`" is treated as `c("1", "2")`).

Following (Wickham 2010), **stringi** makes sure the output data types are consistent and that different functions are interoperable. This makes operation chaining easier and less error prone.

For example, `stri_extract_first_regex()` finds the first occurrence of a pattern in each string, therefore the output is a character of the same length as the input (with recycling rule in place if necessary).

```
R> haystack <- c("bacon", "spam", "jam, spam, bacon, and spam")
R> stri_extract_first_regex(haystack, "\\b\\w{1,4}\\b")

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

Note that a no-match (here, we have been looking for words of at most 4 characters) is marked with a missing string. This makes the output vector size consistent with the length of the inputs.

On the other hand, `stri_extract_all_regex()` identifies all occurrences of a pattern, whose counts may differ from input to input, therefore it yields a list of character vectors.

```
R> stri_extract_all_regex(haystack, "\\b\\w{1,4}\\b", omit_no_match=TRUE)

[[1]]
character(0)

[[2]]
[1] "spam"

[[3]]
[1] "jam"  "spam" "and"  "spam"
```

If the 3rd argument was not specified, a no-match would be represented by a missing value (for consistency with the previous function).

Also, care is taken so that the "data" or "`x`" argument is most often listed as the first one (e.g., in base R we have `grepl(needle, haystack)` vs `stri_detect(haystack, needle)` here). This makes the functions more intuitive to use, but also more forward pipe operator-friendly (either when using "`|>`" introduced in R 4.1 or "`%>%`" from **magrittr**).

Furthermore, for increased convenience, some functions have been added despite the fact that they can be trivially reduced to a series of other calls. In particular, writing:

```
R> stri_sub_all(haystack,
+    stri_locate_all_regex(haystack, "\\b\\w{1,4}\\b", omit_no_match=TRUE))
```

yields the same result as in the previous example, but refers to `haystack` twice.

3.6. Further deviations from base R

stringi can be used as a replacement of the existing string processing functions. Also, it offers many facilities not available in base R. Except for being fully vectorised with respect to all crucial arguments, propagating missing values and empty vectors consistently, and following coherent naming conventions, our functions deviate from their classic counterparts even further.

Following Unicode standards. Thanks to the comprehensive coverage of the most important services provided by **ICU**, its users gain access to collation, pattern searching, normalisation, transliteration, etc., that follow the recent Unicode standards for text processing in any locale. Due to this, as we state in Section 9.2, all inputs are converted to Unicode and outputs are always in UTF-8.

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

For example, let us consider the matching of "all letters" by means of the built-in `gregexpr()` function and the **TRE** (`perl=FALSE`) and **PCRE** (`perl=TRUE`) libraries using a POSIX-like and Unicode-style character set (see Section 6 for more details):

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

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

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

On Windows, when `x` is marked as UTF-8 (see Section 9.2), the author obtained:

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

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

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

As we mention in Section 7, when **stringi** links to **ICU** built from sources (`install.packages("stringi", configure.args="--disable-pkg-config")`), we are always guaranteed to get the same results on every platform.

High performance of stringi. Because of the aforementioned reasons, functions in **stringi** do not refer to their base R counterparts. The operations that do not rely on **ICU** services have been rewritten 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  35.136  37.574  52.371  39.285  54.978 111.37   100
    join3  79.476  84.985  89.886  87.205  92.813 139.65   100
 r_paste2  93.374  98.611 117.285 102.867 116.920 209.74   100
 r_paste3 199.777 211.263 253.166 269.168 281.453 316.77   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   4.9039   5.1316   5.2745   5.2508   5.3741   6.0388   100
   regex 119.5421 124.3005 127.4752 128.1555 129.2322 147.5276   100
    coll 375.7308 391.6492 396.9924 399.2666 404.0569 424.8703   100
   r_tre 129.0912 133.9986 137.2447 137.6145 139.5300 149.9712   100
  r_pcre  73.5176  76.0573  78.3059  78.4355  79.5072  91.2761   100
 r_fixed  44.2047  45.3858  46.8099  46.9502  47.8690  54.1331   100
```

Different default argument and greater configurability. Some functions in **stringi** have different, more natural default arguments, e.g., `paste()` has `sep=" "` but `stri_join()` has `sep=""`. Also, as there is no one-fits-all solution to all problems, many arguments have been introduced for more detailed tuning.

Preserving attributes. Generally, **stringi** preserves no object attributes whatsoever, but a user can make sure themself that this is becomes the case, e.g., by calling "`x[] <- stri_...(x, ...)`" or "`` `attributes<-`(stri_...(x, ...), attributes(x)) ``".

4. Basic string operations

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

4.1. Computing length and width

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

Let us consider the following character vector:

```
R> x <- c("spam", " 你好", "\u200b\u200b\u200b", NA_character_, "")
```

The `x` object consists of 5 character strings:

```
R> length(x)
```

```
[1] 5
```

`stri_length()` computes the *length* of each string. More precisely, the function gives the number of Unicode code points in each string, see Section 9.1 for more details.

```
R> stri_length(x)
```

```
[1]  4  2  3 NA  0
```

The first string carries 4 ASCII (English) letters, the second consists of 2 Chinese characters (U+4F60, U+597D; a greeting), and the third one is comprised of 3 zero-width spaces (U+200B). Note that the 5th element in `x` is an empty string, `""`, hence its length is 0. Moreover, there is a missing (`NA`) value at index 4, therefore the corresponding length is *undefined* as well.

When formatting strings for display (e.g., in a report dynamically generated with `Sweave()` or **knitr**; see Xie 2015), a string's *width* estimate may be more informative – an approximate number of text columns it will occupy when printed using a monospaced font. In particular, many Chinese, Japanese, Korean, and most emoji characters take up two text cells. Some code points, on the other hand, might be of width 0 (e.g., the above ZERO WIDTH SPACE, U+200B).

```
R> stri_width(x)
```

```
[1]  4  4  0 NA  0
```

4.2. Joining

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

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

```
R> x <- c("tasty", "delicious", "yummy", NA)
R> x %s+% " " %s+% c("spam", "bacon")
```

```
[1] "tasty spam"      "delicious bacon" "yummy spam"      NA
```

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

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

```
[1] "tasty, delicious, yummy"
```

Note that the token separator, given by the `collapse` argument, defaults to the empty string.

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

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

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

By default, the `sep` argument, which controls how corresponding strings are delimited, is set to the empty string (like in the base `paste0()` but unlike in `paste()`). Moreover, `collapse` is `NULL`, which means that the resulting outputs will not be joined to form a single string. This can be changed if need be:

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

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

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

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

```
R> stri_dup(letters[1:5], 1:5)  # synonym: letters[1:5] %s*% 1:5
```

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

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=", ")  # collapse=NULL

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

This way, a list of character vectors can be converted to a character vector. Such sequences of variable length sequences of strings are generated by, amongst others, `stri_sub_all()` and `stri_extract_all()`.

4.3. Extracting and replacing substrings

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

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

For example, here is how to select specific elements in a vector:

```
R> x <- c("spam", "buckwheat", "", NA, "bacon")
R> x[1:3]                           # from 1st to 3rd string

[1] "spam"      "buckwheat" ""

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

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

Exclusion of elements at specific positions can be performed like:

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

[1] "buckwheat" ""          NA          "bacon"
```

Filtering based on a logical vector can be used to extract strings fulfilling desired criteria:

```
R> x[!stri_isempty(x) & !is.na(x)]

[1] "spam"      "buckwheat" "bacon"
```

[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.

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"
```

Negative indices count from the end of a string.

```
R> stri_sub(y, -15, length=5)  # 5 code points from 15th last

[1] "bacon"
```

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

```
R> (z <- stri_sub_all(
+                 c("spam",     "bacon", "sorghum"),
+    from   = list(c(1, 3, 4), -3,      c(2, 4)),
+    length = list(1,           3,      c(4, 3))))

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

[[2]]
[1] "con"

[[3]]
[1] "orgh" "ghu"
```

As the number of substrings to extract from each string might vary, the result is a list of character strings. We have obtained: substrings of length 1 starting at positions 1, 3, and 4 in `x[1]`, then a length-3 substring that starts at the 3rd code point from the end of `x[2]`, and length-4 and -3 substrings starting at, respectively, the 2nd and 4th code point of `x[3]` (where `x` denotes the subsetted vector).

"From–to" and "from–length" 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 <- 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"
```

Due to recycling, this has extracted elements at positions 1:2 from the 1st string, at 3:4 from the 2nd one, 5:6 from the 1st, and 7:8 from the 2nd again.

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"
```

This time, we extract four identical sections from each of the two inputs.

Moreover, if the second column of the index matrix is named `"length"` (and only if this is exactly the case), i.e., the indexer is of the form `cbind(from, length=length)`, extraction will be based on the extracted chunk size.

Replacing substrings. `stri_sub_replace()` returns a version of a character vector with some chunks replaced by other strings:

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

[1] "aXcde"  "ABCuvw"
```

The above replaced "`b`" (the length-1 substring starting at index 2 of the 1st string) with "`X`" and "`DE`" (the length-2 substring at index 4 of the 2nd string) with "`uvw`".

Similarly, `stri_sub_replace_all()` replaces multiple substrings within each string in a character vector:

```
R> stri_sub_replace_all(
+                     c("abcde",  "ABCDE"),
+    from        = list(c(2, 4),  c(0,    3,   6)),
+    length      = list(  1,      c(0,    2,   0)),
+    replacement = list(  "Z",    c("uu", "v", "wwww")))

[1] "aZcZe"      "uuABvEwwww"
```

Note how we have obtained the insertion of new content at the start and the end of the 2nd input.

Replacing substrings in-place. The corresponding replacement functions modify a character vector in-place:

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

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

Note that the state of `y` has changed in such a way that the substring of length 3 starting at the 7th code point was replaced by a length-4 content.

Many replacements within a single string are also possible:

```
R> y <- "aa bb cc"
R> stri_sub_all(y, c(1, 4, 7), length=2) <- c("A", "BB", "CCC")
R> print(y)                            # y has changed

[1] "A BB CCC"
```

This has replaced 3 length-2 chunks within `y` with new content.

5. Code-pointwise comparing

There are many circumstances where we are faced with testing whether two strings (or parts thereof) consist of exactly the same Unicode code points, in exactly 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.

5.1. 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, equivalently, the `stri_cmp_eq()` function. Moreover, `%s!==%` and `stri_cmp_neq()` implement the not-equal-to relation.

```
R> "actg" %s===% c("ACTG", "actg", "act", "actga", NA)

[1] FALSE  TRUE FALSE FALSE    NA
```

Due to recycling, the first string was compared against the 5 strings in the 2nd operand. There is only 1 exact match.

5.2. Searching for fixed strings

For detecting if a string contains a given fixed substring (code-pointwisely), the fast KMP (Knuth, Morris, and Pratt 1977) 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).

Table 1 lists the string search functions available in **stringi**. Below we explain their behaviour in the context of fixed pattern matching. Notably, their description is quite detailed, because – as we shall soon find out – the corresponding operations are available for the two other search engines: based on regular expressions and the **ICU** Collator, see Section 6 and Section 7.

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 some replacement strings
`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 1: String search/pattern matching functions in **stringi**. Each function, unless otherwise indicated, can be used in conjunction with any search engine, e.g., we have `stri_count_fixed()` (see Section 5), `stri_detect_regex()` (see Section 6), and `stri_split_coll()` (see Section 7).

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`); available in `stri_extract_all_fixed()`, `stri_locate_all_fixed()`, and `stri_count_fixed()`

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

5.3. Counting matches

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

```
R> stri_count_fixed("abcabcdefabcabcabdc", "abc")  # search pattern is "abc"

[1] 4
```

5.4. Search engine options

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

First, we may switch on the simplistic[3] case-insensitive matching.

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

[1] 3
```

Second, we can indicate our interest 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
```

5.5. Detecting and subsetting patterns

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

[3]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 7 (and partially Section 6) for a workaround.

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

```
R> stri_detect_fixed(x, "abc", 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 2 matching resource IDs".

There are also functions that verify whether a string starts or ends[4] 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

[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", omit_na=TRUE)

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

The above is equivalent to `x[which(stri_detect_fixed(x, "abc"))]` (note the argument responsible for the removal of missing values), but avoids writing `x` twice. It hence is particularly convenient when `x` is generated programmatically on the fly, using some complicated expression. Also, it works well with the forward pipe operator, as we can write "`x |> stri_subset_fixed("abc", omit_na=TRUE)`".

There is also a replacement version of this function:

```
R> stri_subset_fixed(x, "abc") <- c("*****", "***")  # modifies x in-place
R> print(x)  # x has changed
```

[4] 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] "*****" "***"   "def"   "*****" "uabdc" "dab"   NA      "***"
```

5.6. Locating and extracting patterns

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

```
R> x <- c("aga", "actg", NA, "AGagaGAgaga")
R> stri_locate_first_fixed(x, "aga")

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

R> stri_locate_last_fixed(x, "aga", get_length=TRUE)

     start length
[1,]     1      3
[2,]    -1     -1
[3,]    NA     NA
[4,]     9      3
```

In both examples we obtain a two-column matrix with the number of rows determined by the recycling rule (here: the length of `x`). In the former case, we get a "from–to" matrix (`get_length=FALSE`; the default) where missing values correspond to either missing inputs or no-matches. The latter gives a "from–length"-type matrix, where negative lengths correspond to the not-founds.

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

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

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

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

[[3]]
     start end
```

```
[1,]    NA  NA

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

Note again that a no-match is indicated by a single-row matrix with two missing values (or with negative length if `get_length=TRUE`). This behaviour can be changed by setting the `omit_no_match` argument to `TRUE`.

Let us recall that "from–to" and "from–length" matrices of the above kind constitute particularly fine inputs to `stri_sub()` and `stri_sub_all()`. However, if merely the extraction of the matching substrings is needed, it will be more convenient to rely on the functions from the `stri_extract()` family:

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

[1] "aga" NA    NA    "AGa"

R> 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" "aga"
```

5.7. Replacing pattern occurrences

In order to replace each match with a corresponding replacement string, we can refer to `stri_replace_all()`:

```
R> x <- c("aga", "actg", NA, "ggAGAGAgaGAca", "agagagaga")
R> stri_replace_all_fixed(x, "aga", "~", case_insensitive=TRUE)

[1] "~"         "actg"      NA          "gg~G~GAca" "~g~ga"
```

Note that the inputs that are not part of any match are left unchanged. The input object is left unchanged, because it is not a *replacement* function per se.

The operation is vectorised with respect to all the three arguments (*haystack*, *needle*, *replacement string*), with the usual recycling behaviour if necessary. If a different arguments' vectorisation scheme is required, we can 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", "llama"))

[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 llama."

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", "llama"),
+    vectorise_all=FALSE)

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

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 superseded by the 1st replacement string, then the search is repeated for the 2nd needle so as to replace it with the 2nd corresponding replacement string, and so forth.

Moreover, `stri_replace_first()` and `stri_replace_last()` can identify and replace the first and the last match, respectively.

5.8. 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, ",", omit_empty=TRUE)

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

[[2]]
[1] "e"

[[3]]
```

```
character(0)

[[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 is also an option to limit the number of tokens (parameter `n`).

6. Regular expressions

Regular expressions (*regex*es) provide us with a concise grammar for defining systematic patterns which can be sought in character strings. Examples of such patterns include: specific fixed substrings, emojis of any kind, stand-alone sequences of lower-case Latin letters ("words"), substrings that can be interpreted as real numbers (with or without fractional part, also in scientific notation), telephone numbers, email addresses, or URLs.

Theoretically, the concept of regular pattern matching 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 `grep()`, see Table 5):

- ERE[5] (*extended regular expressions* that conform to the POSIX.2-1992 standard); used by default,
- PCRE[6] (*Perl-compatible regular expressions*); activated when `perl=TRUE` is set.

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

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

It is worth noting that most programming languages as well as advanced text editors and IDEs (including **Kate**, **Eclipse**, **VSCode**, and **RStudio**) support 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 `https://unicode-org.github.io/icu/userguide/strings/regexp.html`.

[5] Via the **TRE** library (`https://github.com/laurikari/tre/`).
[6] 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 5) 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 8.1), 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 3: Options for the regular expressions search engine, see `stri_opts_regex()`.

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 6.4 we introduce `stri_match_all_regex()`. Moreover, Table 3 lists the available options for the regex engine.

6.1. Matching individual characters

We begin by discussing different ways to define character sets. In this part, determining the

length of all matching substrings will be quite straightforward.
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> stri_count_regex("spam, eggs, spam, bacon, sausage, and spam", "spam")

[1] 3
```

There are hence 3 occurrences of a pattern that is comprised of 4 code points, "s" followed by "p", then by "a", and ending with "m".
However, this time the case insensitive mode fully supports Unicode matching[7]:

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

[1] TRUE
```

If we wish to include a special character as part of a regular expression – so that it is treated literally – we will need to escape it with a backslash, "\". Yet, the backlash itself has a special meaning to R, see `help("Quotes")`, therefore it needs to be preceded by another backslash.

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

[1] 3
```

In other words, the R string `"\\."` is seen by the regex engine as "`\.`" and interpreted as the dot character (literally). Alternatively, since R 4.0 we can also input the so-called literal strings like `r"(\.)"`.

Matching any character. The (unescaped) dot, "`.`", matches any code point except the newline.

```
R> x <- "Ham, spam,\njam, SPAM, eggs, and spam"
R> stri_extract_all_regex(x, "..am", case_insensitive=TRUE)

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

The above matches non-overlapping length-4 substrings that end with "am".
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> stri_extract_all_regex(x, "..am", dot_all=TRUE, case_insensitive=TRUE)

[[1]]
[1] "spam"  "\njam" "SPAM"  "spam"
```

[7] This does not mean, though, that it considers canonically equivalent strings as equal, see Section 7.2 for a discussion and a workaround.

Defining character sets. Sets of characters can be introduced by enumerating their members within a pair of square brackets. For instance, "`[abc]`" denotes the set $\{\texttt{a}, \texttt{b}, \texttt{c}\}$ – 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] "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 code point except "`a`", "`b`", and "`c`". Here is an example where we seek any substring that consists of 3 non-spaces.

```
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!"
```

Defining Code Point Ranges. Each Unicode code point can be referenced by its unique numeric identifier, see Section 9.1 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> stri_extract_all_regex("In 2020, Gągolewski had fun once.", "[0-9A-Za-z]")

[[1]]
 [1] "I" "n" "2" "0" "2" "0" "G" "g" "o" "l" "e" "w" "s" "k" "i" "h" "a" "d"
[19] "f" "u" "n" "o" "n" "c" "e"
```

The above pattern denotes a union of 3 code ranges: digits and ASCII upper- and lowercase letters.

Nowadays, in the processing of text in natural languages, this notation should rather be avoided. Note the missing "`ą`" (Polish "`a`" with ogonek) in the result.

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

```
R> x <- "aąbßÆAĄB 你 123,.;'! \t-+=[]©←→" „²³¾"
R> p <- c("\\p{L}", "\\p{Ll}", "\\p{Lu}", "\\p{N}", "\\p{P}", "\\p{S}")
R> structure(stri_extract_all_regex(x, p), names=p)

$`\\p{L}`
[1] "a"  "ą"  "b"  "ß"  "Æ"  "A"  "Ą"  "B"  " 你"

$`\\p{Ll}`
[1] "a" "ą" "b" "ß"

$`\\p{Lu}`
[1] "Æ" "A" "Ą" "B"

$`\\p{N}`
[1] "1" "2" "3" "²" "³" "¾"

$`\\p{P}`
 [1] "," "." ";" "'" "!" "-" "[" "]" "”" " " „"

$`\\p{S}`
[1] "+" "=" "©" "←" "→"
```

The above yield a match to: arbitrary letters, lowercase letters, uppercase letters, numbers, punctuation marks, and symbols, respectively.

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

```
R> p <- c("\\w", "\\d", "\\s")
R> structure(stri_extract_all_regex(x, p), names=p)

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

$`\\d`
[1] "1" "2" "3"

$`\\s`
[1] " "  "\t"
```

These give: word characters, decimal digits ("`\p{Nd}`"), and spaces ("`[\t\n\f\r\p{Z}]`"), in this order.

Moreover, e.g., the upper-cased "`\P{category}`" and "`\W`" are 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, because they are generally not well-defined.

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

Base R with **PCRE** yields on the current author's machine:

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

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

However, the details of the characters' belongingness to this class depend on the current locale. Therefore, the reader might obtain different results when calling the above.

ICU, on the other hand, always gives:

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

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

```
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**'s "`[:punct:]`".

6.2. Alternating and grouping subexpressions

The alternation operator, "`|`", matches either its left or its right branch, for instance:

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

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

"`|`" has a very low precedence. Therefore, if we wish to introduce an alternative of *sub*expressions, we need to group them, e.g., between round brackets[8]. For instance, "`(sp|h)am`" matches either "`spam`" or "`ham`".

Also, 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. In particular, "`(al|alga|algae)`" can only match "`al`". To fix this, we can write "`(algae|alga|al))`".

Non-grouping parentheses. Some parenthesised subexpressions – those in which the opening bracket is followed by the question mark – have a distinct meaning. In particular, "`(?#...)`" denotes a free-format comment that is 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_join(
+        "(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 kindly refer the reader to Table 3 again.

6.3. Quantifiers

More often than not, a variable number of instances of the same subexpression needs to be captured or its presence should be made optional. This can be achieved by means of the following quantifiers:

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

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

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

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

```
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)"
```

The first regex is greedy: it matches an opening bracket, then as many characters as possible (including "`)`") that are followed by a closing bracket. The two other patterns terminate as soon as the first closing bracket is found.

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+)?"))

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

[[2]]
[1] "12"         "34.5"       "678.901234" "37"         "629"
```

Here, the first regex matches digits, a dot, and another series of digits. The second one finds digits which are possibly (but not necessarily) followed by a dot and a digit sequence.

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
 17.892   0.000  17.900
```

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

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

6.4. Capture groups and references thereto

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

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

```
R> x <- "name='Sir Launcelot', quest='Seek the Grail', favecolour='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,] "favecolour='blue'"     "favecolour" "blue"
```

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. Also, named capture groups defined like "`(?<name>...)`" are fully supported since version 1.7.1 of our package (for historical notes see Hocking 2019).

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

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

Locating capture group matches. The `capture_groups` attribute in `stri_locate_*_regex` enables us to pinpoint the matches to the parenthesised subexpressions as well:

```
R> stri_locate_all_regex(x, "(?<key>\\w+)='(?<value>.+?)'",
+    capture_groups=TRUE, get_length=TRUE)

[[1]]
     start length
[1,]     1     20
[2,]    23     22
[3,]    47     17
attr(,"capture_groups")
attr(,"capture_groups")$key
     start length
[1,]     1      4
[2,]    23      5
[3,]    47     10

attr(,"capture_groups")$value
     start length
[1,]     7     13
[2,]    30     14
[3,]    59      4
```

Note that each item in the resulting list is equipped with a `"capture_groups"` attribute. For instance, `attr(result[[1]], "capture_groups")[[2]]` extracts the locations of the matches to the 2nd capture group in the first input string.

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`", then "`$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 favecolour"
```

Named capture groups can be referred to too:

```
R> stri_replace_all_regex(x, "(?<key>\\w+)='(?<value>.+?)'",
+    "${value} is a ${key}")

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

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

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

```
R> stri_extract_all_regex("<strong><em>spam</em></strong><code>eggs</code>",
+    c("<[a-z]+>.*?</[a-z]+>", "<([a-z]+)>.*?</\\1>"))

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

[[2]]
[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>. Named capture groups can be referenced using the \k<name> syntax (the angle brackets are part of the token), as in, e.g., "<(?<tagname>[a-z]+)>.*?</\k<tagname>>".

6.5. Anchoring

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

Matching at the beginning or end of a string. "^" and "$" 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", "sausage")
R> p <- c("spam", "^spam", "spam$", "spam$|^spam", "^spam$")
R> structure(outer(x, p, stri_detect_regex), dimnames=list(x, p))

               spam ^spam spam$ spam$|^spam ^spam$
spam egg       TRUE  TRUE FALSE        TRUE  FALSE
bacon spam     TRUE FALSE  TRUE        TRUE  FALSE
spam           TRUE  TRUE  TRUE        TRUE   TRUE
egg spam bacon TRUE FALSE FALSE       FALSE  FALSE
sausage       FALSE FALSE FALSE       FALSE  FALSE
```

The 5 regular expressions match "spam", respectively, anywhere within the string, at the beginning, at the end, at the beginning or end, and in strings that are equal to the pattern itself.

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 following example, we match all stand-alone numbers[9]:

[9] This regular expression is provided for didactic purposes only.

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

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

Note the possessive quantifier, "`?+`": try matching a dot and a sequence of digits, and if it is present but not followed by a word boundary, do not retry by matching a word boundary 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> stri_extract_all_regex("I like spam, spam, eggs, and spam.",
+    c("\\w+(?=[,.])", "\\w++(?![,.])"))

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

[[2]]
[1] "I"    "like" "and"
```

The first regex captures words that end with "`,`" or "`.`". The second one matches words that end neither with "`,`" nor "`.`".

7. Collation

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

The **ICU** Collation Service[10] provides the basis for string comparison activities such as string sorting and searching, or determining if two strings are equivalent. This time, though, due to its conformance to the Unicode Collation Algorithm (Davis, Whistler, and Scherer 2021), we may expect that the generated results will meet the requirements of the culturally correct natural language processing in any *locale*.

7.1. 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.

[10] See the **ICU** User Guide on Collation, https://unicode-org.github.io/icu/userguide/collation/.

Specifying locales. A locale specifier[11] 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 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"` for 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 8.5 for "`@calendar=hebrew`", amongst others).

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

```
R> length(stri_locale_list())
```

```
[1] 784
```

As the number of supported locales is very high, here we shall display only 5 randomly chosen ones:

```
R> sample(stri_locale_list(), 5)
```

```
[1] "nl_CW"      "pt_CH"      "ff_Latn_SL" "en_PH"      "en_HK"
```

Querying for locale-specific services. The availability of locale-specific services can only be determined during the request for a particular resource[12], which 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 macOS, whose users typically download the pre-compiled versions of the package from CRAN. However, on various flavours

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

[12] For more details, see the **ICU** User Guide on Locales, https://unicode-org.github.io/icu/userguide/locale/.

of GNU/Linux and other Unix-based systems, the system **ICU** is used more eagerly[13]. To force building **ICU** from sources, we may call:

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

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

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

As we have stated in the introduction, in this paper we use:

```
R> stri_locale_get()

[1] "en_AU"
```

i.e., the Australian-English locale (which formats dates like "17 July 2021" and uses metric units of measurement).

7.2. Testing string equivalence

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

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

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

[1] TRUE
```

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

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

[1] "Gągolewski" "Gagolewski"

R> stri_duplicated(x)  # from_last=FALSE
```

[13] 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.

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

Moreover, `stri_duplicated_any()` returns the index of the first non-unique element.

7.3. 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, `stri_order()` yields the corresponding (stable) ordering permutation, and `stri_rank()` ranks strings within a vector.

For instance, here is a comparison in the current default locale (Australian-English):

```
R> "chaotic" %s<% "hard"  # c < h
```

```
[1] TRUE
```

Similar comparison in Polish:

```
R> stri_cmp_lt("chłodny", "hardy", locale="pl_PL")  # c < h
```

```
[1] TRUE
```

And now for something completely different – the Slovak language:

```
R> stri_cmp_lt("chladný", "hladný", locale="sk_SK") # ch > h
```

```
[1] FALSE
```

This is an example of the locale-aware comparison that is context-sensitive and which goes beyond the simple code-pointwise comparison. In the example above, a *contraction* occurred: in Slovak, two code points "`ch`" are treated as a single entity and are sorted after "`h`":

Compare the ordering of Polish and Slovak words:

```
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")
```

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 input data are in the FCD ("fast C or D") form; if data are not in the FCD form, the incremental NFD normalisation is performed, see Section 9.4; 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 less than or equal to the variable top value to be ignored on the primary level and moved to the quaternary level; this can be used to, e.g., ignore punctuation, see the examples provided
`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 4: Options for the **ICU** Collator that can be passed to `stri_opts_collator()`.

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

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

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

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

7.4. Collator options

Table 4 lists the options that can be passed to `stri_opts_collator()` via the dot-dot-dot parameter, "`...`", in all the functions that rely on the **ICU** Collator. Below we would like to attract the kind reader's attention to some of them.

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

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

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

[1] TRUE
```

which is not the case at the default strength level (3).

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

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

Ignoring case. Note which strings are deemed equivalent when considering different collation strengths:

```
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=2)                  # ß != ss, case insensitive

[1] "groß"  "gross"
```

Hence, strength equal to 1 takes only primary differences into account. Strength of 2 will also be sensitive to secondary differences (distinguishes between "ß" and "ss" above), but will ignore tertiary differences (case).

Also, introducing an extra case level yields a case sensitive comparison that ignores secondary differences:

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

[1] "groß"  "GROSS" "Groß"
```

Ignoring some punctuation. Here are some effects of changing the `alternate_shifted` option, which allows for ignoring some punctuation marks:

```
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"
```

In the latter case, all strings are considered equivalent. Ignoring case but not punctuation yields:

```
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. By default, just like in base R and most other programming languages, a lexicographic ordering is used: the corresponding code points are being compared one by one, from left to right, and once a difference is detected, the result is returned immediately.

```
R> x <- c("a1", "a2", "a11", "a1", "a99", "a10", "a100", "a2", "a9", "a2")
R> stri_sort(x)

 [1] "a1"   "a1"   "a10"  "a100" "a11"  "a2"   "a2"   "a2"   "a9"   "a99"
```

For example, `"a99"` is ordered after `"a100"`, because `"a" == "a"` (first characters are equal) but then `"9" > "1"` (second characters are already different).

Let us, however, note the effect of setting the `numeric` option on the sorting of strings that involves numbers:

```
R> stri_sort(x, numeric=TRUE)

 [1] "a1"   "a1"   "a2"   "a2"   "a2"   "a9"   "a10"  "a11"  "a99"  "a100"
```

Here is an example of sorting a data frame with respect to two criteria:

```
R> X <- data.frame(a=x, b=runif(length(x)))
R> X[order(-stri_rank(X$a, numeric=TRUE), X$b), ]

      a        b
7  a100 0.528105
5   a99 0.940467
3   a11 0.408977
6   a10 0.045556
9    a9 0.551435
10   a2 0.456615
2    a2 0.788305
8    a2 0.892419
1    a1 0.287578
4    a1 0.883017
```

The object is now ordered by the first column decreasingly (using a "numeric" order) and ties are resolved based on increasing values in the second column.

A note on compatibility equivalence. In Section 9.4 we describe different ways to normalise canonically equivalent code point sequences so that they are represented by the same code points, which can account for some negligible differences (as in the "a with ogonek" example above).

Apart from ignoring punctuation and case, the Unicode Standard Annex #15 (Davis and Whistler 2021) also discusses the so-called *compatibility* equivalence of strings. This is a looser form of similarity; it is observed where there is the same *abstract* content, yet displayed by means of different glyphs, for instance "¼" (U+00BC) vs "`1/4`" or "ℝ" vs "R". In the latter case, whether these should be treated as equal, depends on the context (e.g., this can be the set of real numbers vs one's favourite programming language). Compatibility decompositions (NFKC, NFKD) mentioned in Section 9.4 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.

7.5. Searching for fixed strings revisited

The **ICU** Collator can also be utilised where there is a need to locate the occurrences of simple textual patterns. The counterparts of the string search functions described in Section 5 have their names ending with `*_coll()`. They are slower than them, but are more appropriate in NLP activities.

For instance:

```
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
```

8. Other operations

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

8.1. Analysing text boundaries

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

Generally, text boundary analysis is a locale-sensitive operation, see (Davis and Chapman 2021). For example, in Japanese and Chinese, spaces are not used for separation 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[14] comes in four flavours (see the `type` option in `stri_opts_brkiter()`): `character`, `work`, `line_break`, and `sentence`.

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

```
R> x <- "The\u00a0above-mentioned    features are useful. " %s+%
+    "My hovercraft is full of eels, eggs, and spam."
```

Number of sentences:

```
R> stri_count_boundaries(x, type="sentence")

[1] 2
```

The list of all the words:

```
R> stri_extract_all_words(x)
```

[14] See the **ICU** User Guide on Boundary Analysis, https://unicode-org.github.io/icu/userguide/boundaryanalysis/.

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

8.2. Trimming, padding, and other formatting

The following functions can be used for pretty-printing character strings or text on the console, dynamically generating reports (e.g., with `Sweave()` or **knitr**; see Xie 2015), or creating text files (e.g., with `stri_write_lines()`; see Section 9.3).

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

```
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."
```

Word wrapping. The `stri_wrap()` function splits each (possibly long) string in a character vector into chunks of at most a given width. 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=74, indent=8, exdent=4, 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.
```

Applying string templates. `stri_sprintf()` is a Unicode-aware rewrite of the built-in `sprintf()` function. In particular, it enables formatting and padding based on character width, not just the number of code points. The function is also available as a binary operator `%s$%`, which is similar to Python's `%` overloaded for objects of type `str`.

```
R> cat(stri_sprintf("[%6s]", c("abcd", "\u200b\u200b\u200bąß²€")), sep="\n")

[  abcd]
[  ąß²€]
```

The above guarantees that the two output strings are of at least width of 6 (plus the square brackets).

8.3. 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.

For example, here are 5 random ACTG strings of lengths from 2 to 6:

```
R> stri_rand_strings(5, 2:6, "[ACTG]")

[1] "CT"     "CTT"    "AGTG"   "CTCGG"  "ATAACT"
```

See Section 6.1 and `help("stringi-search-charclass")` for different ways to specify character sets.

8.4. Transliterating

Transliteration, in its broad sense, deals with the substitution of characters or their groups for different ones, according to some well-defined, possibly context-aware, 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. When a fast 1-to-1 code point translation 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. `stri_trans_general()` provides access to a wide range of text transforms defined by **ICU**, whose catalogue can be accessed by calling `stri_trans_list()`.

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

[1] "Any-und_FONIPA" "Any-FCD"        "Deva-Guru"      "yo-yo_BJ"
[5] "Taml-Orya"      "Tamil-Arabic"   "hy-ar"          "de-ASCII"
[9] "Any-Kana"
```

For example, below we apply a transliteration chain: first, we convert to upper case, and then we convert characters in the Latin script to ASCII.

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

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

Custom rule-based transliteration is also supported[15]. It can be used, for instance, to generate different romanisations of non-Latin alphabets.

8.5. Parsing and formatting date and time

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

```
R> x <- "1 maja 2021 r., godz. 17:17:32"
```

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

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

[1] "2021-05-01 17:17:32 CEST"
```

[15] See the **ICU** User Guide on General Transforms for more details, https://unicode-org.github.io/icu/userguide/transforms/general/.

This function returns an object of class `POSIXct`, for compatibility with base R. Note, however, that **ICU** uses its own format patterns[16]. For convenience, `strftime()`- and `strptime()`-like templates can be converted with `stri_datetime_fstr()`:

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

[1] "2021-05-01 17:17:32 CEST"
```

For example, here is how we can access different calendars:

```
R> stri_datetime_format(
+    stri_datetime_create(2020, 1:12, 1),
+    "date_long",
+    locale="@calendar=hebrew")

 [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")

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

Above we have selected the Hebrew calendar within the English locale and the Japanese calendar in the Japanese locale.

9. Input and output

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

9.1. 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 is the number of the code points in some popular categories (compare Section 6.1), such as letters, numbers, and the like.

[16] See the **ICU** User Guide on Formatting Dates and Times, https://unicode-org.github.io/icu/userguide/format_parse/datetime/.

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

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

Yet, most of the code points are still unallocated. The Unicode standard is occasionally updated, e.g., the most recent versions were supplemented with over 1,000 emojis.

The first 255 code points are identical to the ones defined by ISO/IEC 8859-1 (ISO Latin-1; "Western European"), which itself extends US-ASCII (codes $\leq 127 = 7\mathrm{F}_{16}$). 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 escape it with another backslash).

```
R> x <- "zß 你好"
R> stri_escape_unicode(x)

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

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}"
```

9.2. 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]]  # little-endian

 [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 from files (e.g., when `scan()` is called). For instance, Central European versions of Windows will assume the "`windows-1250`" code page. MacOS as well as most Linux boxes work with UTF-8 by default[17]

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 function in **stringi**, 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.

9.3. Reading and writing text files and converting between encodings

According to a report by W3Techs[19], as of 2021–05–05, 96.8% of websites use UTF-8. Nevertheless, other encodings can still be encountered.

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

For instance, below we read a text file encoded in ISO-8859-1:

```
R> x <- stri_read_lines("ES_latin1.txt", encoding="ISO-8859-1")
R> head(x, 4)  # x is in UTF-8 now
```

[17] It is expected that future R releases will support UTF-8 natively thanks to the Universal C Runtime (UCRT) that is available for Windows 10.

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

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

```
[1] "CANTO DE CALÍOPE - Miguel de Cervantes"
[2] ""
[3] "Al dulce son de mi templada lira,"
[4] "prestad, pastores, el oído atento:"
```

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, 24)  # vector of type raw
```

```
 [1] 43 41 4e 54 4f 20 44 45 20 43 41 4c cd 4f 50 45 20 2d 20 4d 69 67 75 65
```

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       0.81
2 ISO-8859-2       ro       0.36
3 ISO-8859-9       tr       0.20
4   UTF-16BE                0.10
5   UTF-16LE                0.10
```

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

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

```
R> y <- stri_encode(x, from="ISO-8859-1", to="UTF-8")
```

`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"`.

Splitting the output into text lines gives:

```
R> tail(stri_split_lines1(y), 4)  # spoiler alert!

[1] "A Homero iguala si a escrebir intenta,"
[2] "y a tanto llega de su pluma el vuelo,"
[3] "cuanto es verdad que a todos es notorio"
[4] "el alto ingenio de don DIEGO OSORIO."
```

9.4. Normalising strings

In Section 7.2 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 2021).

```
R> x <- "a\u0328 ą"   # a, combining ogonek, space, a with ogonek
R> stri_enc_toutf32(  # code points as decimals
+    c(x, stri_trans_nfc(x), stri_trans_nfd(x)))

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

[[2]]
[1] 261  32 261

[[3]]
[1]  97 808  32  97 808
```

Above we see some example code points before, after NFC, and after NFD normalisation, respectively.

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

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

```
R> stri_trans_nfkd("r²⁀")

[1] "r2{"
```

Such text might be much easier to process or analyse statistically.

10. Closing remarks

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

Users seeking Unicode-aware replacements for base R string processing functions are kindly referred to the **stringx** package (Gagolewski 2021), which is a set of wrappers around **stringi** offering a more classic API (functions such as `grepl()`, `substring()`, etc., compare Table 5).

stringi functions can also be accessed from within C++ code. Authors of statistical/data analysis software who would like to speed up their projects are encouraged to check out the **ExampleRcppStringi** package available at https://github.com/gagolews/ExampleRcppStringi, which serves as a working prototype developed using **Rcpp** (Eddelbuettel 2013).

Future of stringi. Over the years, many useful R packages related to text processing have been developed, see (Feinerer, Hornik, and Meyer 2008; Welbers, Van Atteveldt, and Benoit 2017) for some reviews. Several of them are listed in the CRAN Task View on Natural Language Processing[20]. At the time of the writing of this paper, **stringi** itself had over 200 strong (direct) reverse dependencies and has established itself as one of the most frequently downloaded R extension. Its user base is growing steadily.

Most importantly, the package can be relied upon by other software projects as its API is considered stable and most changes are backward compatible.

Future work will involve the porting of **stringi** to different scientific/statistical computing environments, including Julia and Python with the **NumPy** (van der Walt, Colbert, and Varoquaux 2011) ecosystem, offering more Unicode-aware alternatives to the vectorised text processing facilities from `numpy.char` and **pandas** (McKinney 2017, Chap. 7).

Moreover, further extensions of **stringi** shall be conveyed in order to provide an even broader coverage of **ICU** services.

Acknowledgements

stringi is an open source project distributed under the terms of the BSD-3-clause license. Its most recent development snapshot is available through GitHub at https://github.com/gagolews/stringi. The bug- and feature request tracker can be accessed from https:

[20] See https://cran.r-project.org/web/views/NaturalLanguageProcessing.html.

//github.com/gagolews/stringi/issues. Moreover, its homepage – which includes a reference manual documents the package's API in detail – is located at https://stringi.gagolewski.com/.

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

References

Chambers J (2008). *Software for Data Analysis. Programming with R*. Springer-Verlag.

Clayden J (2019). ***ore**: An R Interface to the **Onigmo** Regular Expression Library*. R package version 1.6.3, URL https://CRAN.R-project.org/package=ore.

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

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

Davis M, Chapman C (2021). "Unicode Standard Annex #29: Unicode Text Segmentation." URL https://unicode.org/reports/tr29/.

Davis M, Heninger A (2021). "Unicode Technical Standard #18: Unicode Regular Expressions." URL https://www.unicode.org/reports/tr18/.

Davis M, Whistler K (2021). "Unicode Standard Annex #15: Unicode Normalization Forms." URL https://www.unicode.org/reports/tr15/.

Davis M, Whistler K, Scherer M (2021). "Unicode Technical Standard #10: Unicode Collation Algorithm." URL https://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 J (2006). *Mastering Regular Expressions*. O'Reilly.

Gagolewski M (2021). ***stringx**: Drop-in Replacements for Base R String Functions Powered by **stringi***. R package version 0.1.1, URL https://stringx.gagolewski.com/.

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

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

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

Kleene SC (1951). "Representation of Events in Nerve Nets and Finite Automata." *Technical Report RM-704*, The RAND Corporation, Santa Monica, CA. URL https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf.

Knuth D, Morris JH, Pratt V (1977). "Fast Pattern Matching in Strings." *SIAM Journal on Computing*, **6**(2), 323–350.

Knuth D, Plass M (1981). "Breaking Paragraphs into Lines." *Software: Practice and Experience*, **11**, 1119–1184.

Kurtz S, *et al.* (2004). "Versatile and Open Software for Comparing Large Genomes." *Genome Biology*, **5**, R12.

McKinney W (2017). *Python for Data Analysis*. O'Reilly.

Rabin M, Scott D (1959). "Finite Automata and Their Decision Problems." *IBM Journal of Research and Development*, **3**, 114–125.

R Development Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org.

Ritchie DM, Thompson KL (1970). "**QED** Text Editor." *Technical Report 70107-002*, Bell Telephone Laboratories, Inc. URL https://wayback.archive-it.org/all/20150203071645/http://cm.bell-labs.com/cm/cs/who/dmr/qedman.pdf.

Szpankowski W (2001). *Average Case Analysis of Algorithms on Sequences*. John Wiley & Sons.

van der Loo M (2014). "The **stringdist** Package for Approximate String Matching." *The R Journal*, **6**(1), 111–122.

van der Loo M, de Jonge E (2018). *Statistical Data Cleaning with Applications in R*. John Wiley & Sons.

van der Walt S, Colbert SC, Varoquaux G (2011). "The **NumPy** Array: A Structure for Efficient Numerical Computation." *Computing in Science Engineering*, **13**(2), 22–30.

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

Wenfeng Q (2020). ***re2r**: **RE2** Regular Expression*. R package version 1.0.0, URL https://github.com/qinwf/re2r.

Whistler K, Iancu L (2021). "Unicode Standard Annex #44: Unicode Character Database." URL https://unicode.org/reports/tr44/.

Wickham H (2010). "**stringr**: Modern, Consistent String Processing." *The R Journal*, **2**(2), 38–40.

Wickham H (2021). ***rvest**: Easily Harvest (Scrape) Web Pages*. R package version 1.0.0, URL https://CRAN.R-project.org/package=rvest.

Wickham H, Grolemund G (2017). *R for Data Science*. O'Reilly.

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

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

Affiliation:

Marek Gagolewski
School of Information Technology
Deakin University
Geelong, VIC 3220, Australia
and
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6, 01-447 Warsaw, Poland
E-mail: m.gagolewski@deakin.edu.au
URL: https://www.gagolewski.com/

stringr 0.6.2	**Base R 4.1**	**Purpose**
`str_c()`	`paste()`, `paste0()`, also `sprintf()`	join strings
`str_count()`	`gregexpr()`	count pattern matches
`str_detect()`	`grepl()`	detect pattern matches
`str_dup()`	`strrep()`	duplicate strings
`str_extract()`, `str_extract_all()`	`regmatches()` with `regexpr()`, `gregexpr()`	extract (first, all) pattern matches
`str_length()`	`nchar()`	compute string length
	`nchar(type="width")`	compute string width
`str_locate()`, `str_locate_all()`	`regexpr()`, `gregexpr()`	locate (first, all) pattern matches
`str_match()`, `str_match_all()`	`regmatches()` with `regexec()`, `gregexec()`	extract (first, all) matches to regex capture groups
`str_pad()`		add whitespaces at beginning or end
`str_replace()`, `str_replace_all()`	`sub()`, `gsub()`	replace (first, all) pattern matches with a replacement string
`str_split()`, `str_split_fixed()`	`strsplit()`	split up a string into pieces
`str_sub()`	`substr()`, `substring()`	extract or replace substrings
`str_trim()`	`trimws()`	remove whitespaces from beginning or end
`str_wrap()`	`strwrap()`	split strings into text lines
`word()`		extract words from a sentence
	`startsWith()`, `endsWith()`	determine if strings start or end with a pattern match
	`tolower()`, `toupper()`	case mapping and folding
	`chartr()`	transliteration
	`sprintf()`	string formatting
	`strftime()`, `strptime()`	date-time formatting and parsing

Table 5: Functions in (the historical) **stringr** 0.6.2 and their counterparts in base R 4.1.