

Module 4 (Character string processing)

Contents

1	Representation of character strings	2
1.1	Introduction	2
1.2	Character encoding	2
1.2.1	ASCII	2
1.2.2	8-bit encodings	4
1.2.3	Unicode	5
1.3	Unicode character database	7
1.3.1	General Category Properties	8
1.3.2	Binary properties	8
1.4	Summary	8
1.5	Bibliography	8
2	String processing and searching	10
2.1	Introduction	10
2.2	Basic string processing tasks	11
2.3	Transliteration	13
2.4	Locale-dependent operations	13
2.5	String searching	14
2.6	Summary	17
2.7	Bibliography	17
3	Regular expressions	17
3.1	Introduction	17
3.2	Matching single characters	17
3.3	Matching single characters	18
3.4	Alternation	18
3.5	Quantifiers	19
3.6	Capturing groups and backreferences	20
3.7	Anchors, look-aheads, and look-behinds	21
3.8	Summary	23
3.9	Bibliography	23
4	Date and time	23
4.1	Introduction	23
4.2	Dealing with dates	24
4.3	Dealing with date/time objects	24
4.4	Date/time arithmetic	26
4.5	Summary	26
4.6	Bibliography	27

1 Representation of character strings

1.1 Introduction

R character vectors consist of **character strings**. Each string is a **sequence of bytes**. Recall that the byte is a small integer $\in \{0, 1, \dots, 255\}$. Thus, a character vector is a sequence of byte sequences.

In this part we will discuss:

- How are characters encoded?
- What are the challenges of string processing?

1.2 Character encoding

A **character encoding** (character set, character map, code page) is a mapping of a set of characters (letters, digits, etc.) to another domain (e.g. digits, specific symbol sequences, etc.). Of course, we are interested in representing text in computers.

1.2.1 ASCII

In **8-bit character encodings**, up to 256 characters may be represented: they are mapped to $\{0, 1, \dots, 255\}$. Most modern character-encoding schemes are based on **ASCII**, The American Standard Code for Information Interchange, also known as US-ASCII, see Tab. 1–3. ASCII is based on the English alphabet and encodes 128 characters (codes $\in \{0, 1, \dots, 127\}$).

Table 1: ASCII table, part I

DEC	HEX	Char	DEC	HEX	Char
0	00	Null (\0)	16	10	
1	01		17	11	
2	02		18	12	
3	03		19	13	
4	04		20	14	
5	05		21	15	
6	06		22	16	
7	07	BEL (\a)	23	17	
8	08	BSP (\b)	24	18	
9	09	TAB (\t)	25	19	
10	0A	LF (\n)	26	1A	SUBST
11	0B		27	1B	
12	0C		28	1C	
13	0D	CR (\r)	29	1D	
14	0E		30	1E	
15	0F		31	1F	

Note the following:

```
test <- "R programming 2014"
charToRaw(test) # printed in HEX
## [1] 52 20 70 72 6f 67 72 61 6d 6d 69 6e 67 20 32 30 31 34
as.integer(charToRaw(test)) # now in DEC
## [1] 82 32 112 114 111 103 114 97 109 109 105 110 103 32 50 48 49
## [18] 52
```

In fact, each string processing task is like playing with numeric vectors. For example, here is a way to subset and reverse an ASCII string:

```
test <- "R programming 2014"
rawToChar(rev(charToRaw(test)[3:13]))
```


Table 2: ASCII table, part II

DEC	HEX	Char	DEC	HEX	Char	DEC	HEX	Char
32	20	Space	48	30	0	64	40	@
33	21	!	49	31	1	65	41	A
34	22	"	50	32	2	66	42	B
35	23	#	51	33	3	67	43	C
36	24	\$	52	34	4	68	44	D
37	25	%	53	35	5	69	45	E
38	26	&	54	36	6	70	46	F
39	27	'	55	37	7	71	47	G
40	28	(56	38	8	72	48	H
41	29)	57	39	9	73	49	I
42	2A	*	58	3A	:	74	4A	J
43	2B	+	59	3B	;	75	4B	K
44	2C	,	60	3C	<	76	4C	L
45	2D	-	61	3D	=	77	4D	M
46	2E	.	62	3E	>	78	4E	N
47	2F	/	63	3F	?	79	4F	O


Table 3: ASCII table, part III

DEC	HEX	Char	DEC	HEX	Char	DEC	HEX	Char
80	50	P	96	60	'	112	70	p
81	51	Q	97	61	a	113	71	q
82	52	R	98	62	b	114	72	r
83	53	S	99	63	c	115	73	s
84	54	T	100	64	d	116	74	t
85	55	U	101	65	e	117	75	u
86	56	V	102	66	f	118	76	v
87	57	W	103	67	g	119	77	w
88	58	X	104	68	h	120	78	x
89	59	Y	105	69	i	121	79	y
90	5A	Z	106	6A	j	122	7A	z
91	5B	[107	6B	k	123	7B	{
92	5C	\	108	6C	l	124	7C	
93	5D]	109	6D	m	125	7D	}
94	5E	^	110	6E	n	126	7E	~
95	5F	_	111	6F	o	127	7F	

```
## [1] "gnimmargorp"
```

 **Exercise:** One of the simplest encryption techniques is a *Cæsar's cipher* with shift **h**. It substitutes each *i*-th letter of the English alphabet with the $(i + h)$ -th one (possibly with a kind of “recycling”). Write a function `caesar()` to encrypt a given ASCII string with a **h**-shift *Cæsar's cipher*, where **h** is a given integer.

```
caesar(c("abcABÇąšćxyzXYZ0123!@$$", "She looks nice today."), 1)
## [1] "bcdBCDąšćzyaYZA0123!@$$" "Tif mpplt ojdf upebz."
caesar("V ybir guvf pbhefr!", -13) # decipher
## [1] "I love this course!"
```

 **Exercise:** *Cæsar's cipher* is a type of substitution cipher, which is of course very easy to break. Could you implement some more sophisticated version of this algorithm? Give its implementation to your friend and ask him/her to break it (without looking at the source code). This should not take him/her more than a few minutes.

1.2.2 8-bit encodings

What about the representation of **non-English alphabets**? 8-bit encodings are perhaps the most convenient. Here, each character is represented by 1 byte. Thus, it is easy to:

- Find the number of code points in a string. ($O(1)$ if we know the number of bytes)
- Find the i th character. ($O(1)$)
- Subset a string from i th to j th character.

This storage method is also memory efficient. Among popular encodings we find:

- Polish, Czech, Slovak, Hungarian: ISO-8859-2 (latin2), Windows-1250 (cp1250)
- German, Norwegian, Italian, Portuguese, Spanish, Catalan: ISO-8859-1 (latin1), Windows-1252
- Turkish: ISO-8859-9, Windows-1254
- Latvian, Estonian, Lithuanian: ISO-8859-4, Windows-1257
- Cyrillic: ISO-8859-5, Windows-1251
- Greek: ISO-8859-7, Windows-1253
- Hebrew: ISO-8859-8, Windows-1255
- Arabic: ISO-8859-6, Windows-1256

All of these are **supersets of ASCII**.

A string is a sequence of bytes – how should we interpret it? For example, here are the codes of Polish letters in latin2 and cp1250:

	Ą	Ć	Ę	Ł	Ń	Ó	Ś	Ż	Ź
ISO-8859-2	161	198	202	163	209	211	166	172	175
Windows-1250	165	198	202	163	209	211	140	143	175

	ą	ć	ę	ł	ń	ó	ś	ż	ź
ISO-8859-2	177	230	234	179	241	243	182	188	191
Windows-1250	185	230	234	179	241	243	156	159	191

Note that there are notable differences between some of the codes.

A character conversion example:

```
library("stringi") # we will be using this package extensively
as.integer(stri_conv("ąśćżźółńę", "", "latin2", to_raw=TRUE)[[1]])
## [1] 177 182 230 191 188 243 179 241 234
as.integer(stri_conv("ąśćżźółńę", "", "cp1250", to_raw=TRUE)[[1]])
## [1] 185 156 230 191 159 243 179 241 234
```

Here "" denotes the system's default encoding (see below).

Another example – one byte sequence – different interpretations:

```
x <- as.raw(192:207)
stri_conv(x, "latin1", "")
## [1] "ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎ"
stri_conv(x, "latin2", "")
## [1] "ŘÁÂÃÄĹĆČĚĚĚİİĎ"
```

Check out also e.g. windows-1253 and windows-1256.

Moreover, here is a piece of text in cp1250 interpreted as latin2:

Tekst w CP-1250 interpretowany przy użyciu ISO-8859-2.

```
-- Wełł tś zółłś kredkś ælad mi tu narysuj, Muńku!
-- Cóż, robię co mogę, żółć mnie już zalewa!
```

And now a piece of text in latin2 interpreted as cp1250:

Tekst w ISO-8859-2 interpretowany przy użyciu CP-1250.

```
-- Weł t± żółt± kredk± łlad mi tu narysuj, Muńku!  
-- Cóż, robię co mogę, żółć mnie już zalewa!
```

The original text was **Weź tą żółtą kredką ślad...** Thus, if we apply an incorrect encoding, our text will not be readable. If the encoding is not given and if we know at least a text's language, we can use some statistics and heuristics to guess the encoding. However, such an operation is, at best, imprecise.

Here is an encoding auto-detection test ("Powrót taty" by A. Mickiewicz (a Polish poem), windows-1250-encoded):

```
test <- stri_read_raw("test-cp1250.txt") # read file contents  
## Error: file.exists(fname) is not TRUE  
head(test)  
## [1] "R programming 2014"  
stri_enc_detect(test)  
## [[1]]  
## [[1]]$Encoding  
## [1] "ISO-8859-1" "ISO-8859-2" "UTF-8"      "Shift_JIS"  "GB18030"  
## [6] "EUC-JP"      "EUC-KR"      "Big5"  
##  
## [[1]]$Language  
## [1] "en" "cs" "" "ja" "zh" "ja" "ko" "zh"  
##  
## [[1]]$Confidence  
## [1] 0.6 0.4 0.1 0.1 0.1 0.1 0.1 0.1
```

In this case, the auto-detection works correctly.

1.2.3 Unicode

8-bit encodings are not ideal: recall that 1 byte enables us to represent only 256 possible values. What about very “complex” languages, e.g. Chinese, Japanese, Korean, etc., cf. Tab. 4?

*For historical reasons, international text is often encoded using a language or country dependent character encoding. **With the advent of the internet and the frequent exchange of text across countries** – even the viewing of a web page from a foreign country is a “text exchange” in this context – conversions between these encodings have become important. They have also become a problem, because many characters which are present in one encoding are absent in many other encodings.*

*To solve this mess, the **Unicode encoding** has been created. It is a **super-encoding of all others** and is therefore the default encoding for new text formats like XML*

Source: www.gnu.org/software/libiconv/

The Unicode standard defines > 1,000,000 code points. It of course does not mean that your computer can display them all. **UTF-8** is perhaps the most popular encoding in the Unicode family. Here, each code point is encoded using **1–4 bytes**. Thus, it is a variable-width encoding.

First code point	Last code point	Bytes	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	1	0xxxxxxx			
U+0080	U+07FF	2	110xxxxx	10xxxxxx		
U+0800	U+FFFF	3	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	U+1FFFFF	4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

On my system, UTF-8 is the default encoding. By the way, that is most often the default one on Linux and OS X. In other words, if I input "aą", R marks it as a UTF-8 string. Thus:

Table 4: Top 20 languages according to the number of speakers

Rank	Language	Primary Country	Total Countries	Speakers (millions)
1	Chinese	China	33	1 197
2	Spanish	Spain	31	406
3	English	UK	101	335
4	Hindi	India	4	260
5	Arabic	Saudi Arabia	59	223
6	Portuguese	Portugal	11	202
7	Bengali	Bangladesh	4	193
8	Russian	Russian Fed.	16	162
9	Japanese	Japan	3	122
10	Javanese	Indonesia	3	84
11	German	Germany	18	84
12	Lahnda	Pakistan	7	83
13	Telugu	India	2	74
14	Marathi	India	1	72
15	Tamil	India	6	69
16	French	France	51	69
17	Vietnamese	Viet Nam	3	68
18	Korean	South Korea	6	66
19	Urdu	Pakistan	6	63
20	Italian	Italy	10	61


<http://www.ethnologue.com/statistics/size>


```

stri_length("aa") # how many code points?
## [1] 2
stri_numbytes("aa") # how many bytes are used?
## [1] 3
charToRaw("aa") # bytes
## [1] 61 c4 85
stri_enc_toutf32("aa")[[1]] # 1 code point -> 1 integer
## [1] 97 261

```

There are some limitations of UTF-8. For example, determining the length of a UTF-8 string is $O(n)$. . . We could use UTF-32 (1 code point – one 4-byte integer) to represent a code point, but it is memory inefficient. . . Unfortunately, there is no perfect encoding. But UTF-8 is a good compromise.

 **Exercise:** Write a function `chartable()`, which counts the number of occurrences of each code point in a given UTF-8 encoded string. The result should be stored in a named numeric vector sorted w.r.t. the number of occurrences.

 **Exercise:** A palindrome is a sequence of code points that reads the same forward or reversed. Write a function to determine whether a given string is a palindrome. Some examples: "Madam, I'm Adam", "Never Odd or Even", etc. The most basic implementation may assume that you are given a string consisting of only small letters.

Not each byte stream is valid UTF-8. However, statistically speaking, if a (quite long) sequence of bytes is valid UTF-8, then it is most probably in UTF-8. Moreover, note that each ASCII text is a valid UTF-8 byte sequence

```
test <- stri_read_raw("test-utf8.txt")
## Error: file.exists(fname) is not TRUE
head(test)
## [1] "R programming 2014"
stri_enc_isutf8(test)
## [1] TRUE
stri_enc_isascii(test) # all bytes <= 127?
## [1] TRUE
```

By the way, a UTF-8 text is easy to recognize. Here is a piece of text in UTF-8 interpreted as latin2:

Tekst w UTF-8 interpretowany przy użyciu ISO-8859-2.

```
-- Wełś tÄ... ŁżÄłŁ,tÄ... kredkÄ... Łłlad mi tu narysuj, Muł„ku!
-- CÄłŁż, robiÄ™ co mogÄ™, ŁżÄłŁ,Ä‡ mnie jułż zalewa!
```

And a piece of text in UTF-8 interpreted as cp1250:

Tekst w UTF-8 interpretowany przy użyciu CP-1250.

```
-- Wełś tÄ... ŁŁÄłŁ,tÄ... kredkÄ... Łłlad mi tu narysuj, Muł„ku!
-- CÄłŁŁ, robiÄ™ co mogÄ™, ŁŁÄłŁ,Ä‡ mnie jułŁ zalewa!
```

Some characters in UTF-8 may have ambiguous representations. For example, the Polish character “ą” (a with ogonek, U+0105) may also be represented as “a” (U+0061) and “ogonek” (U+0328). Unfortunately, base R does not know that they are the same characters:

```
"\u0105" == "a\u0328" # a with ogonek, a and ogonek
## [1] FALSE
```

Some word processors and web browsers may also have problems with them: "ą"."ą" "ą" "a," "ą" "ą"

The process of transforming a Unicode code point sequence into an unambiguous form is called **normalization**. Unicode non-normalized text occurs very rarely in practice (e.g. on the internet). However, I feel obliged to note at least the **Unicode normalization form C (NFC)** here:

```
stri_enc_toutf32("\u0105")[[1]]
## [1] 261
stri_enc_toutf32("a\u0328")[[1]]
## [1] 97 808
stri_enc_toutf32(stri_trans_nfc("a\u0328"))[[1]]
## [1] 261
```

Read more about NFC, NFKC, NFD, NFKD, NFKC-casefold: ?stri_trans_nfc. By the way, `stri_trans_nfkc("\ufdfa")`: 1 code point is converted to 18 code points.

1.3 Unicode character database

The Unicode standard is very rich. It is frequently updated, e.g Unicode 7.0 was released on 2014-06-16. See unicode.org, in particular:

- Unicode Technical Standard #10: Unicode Collation Algorithm
- Unicode Standard Annex #15: Unicode Normalization Forms
- Unicode Technical Report #17: Unicode Character Encoding Model
- Unicode Technical Standard #18: Unicode Regular Expressions
- Unicode Standard Annex #44: Unicode Character Database

Text processing software libraries should (but often do not) track and reflect changes in the Unicode standards. Currently only IBM's ICU provides a sufficient level of Unicode conformance.

For example, UCD defines a name for each Unicode code point, see Unicode Standard Annex #34: Unicode Named Character Sequences.

```
stri_trans_general(c("a\u0328", ".", "ą", "@", "1", "!"), "Name")
## [1] "\\N{LATIN SMALL LETTER A}\\N{COMBINING OGONEK}"
## [2] "\\N{FULL STOP}"
## [3] "\\N{LATIN SMALL LETTER A WITH OGONEK}"
## [4] "\\N{COMMERCIAL AT}"
## [5] "\\N{DIGIT ONE}"
## [6] "\\N{EXCLAMATION MARK}"
```

See e.g. unicode.org/cldr/utility/character.jsp?a=0105 for full UCD information on U+0105.

1.3.1 General Category Properties

Each Unicode character is assigned exactly one category, see Table 5. General Categories may be used to classify code points. Moreover, they are very often utilized in string searching tasks.

For example, we may extract all lowercase letters:

```
stri_extract_all_charclass("a,ą@ 1!", "\\p{Ll}")
## [[1]]
## [1] "a" "ą"
```

or extract all digits:

```
stri_extract_all_charclass("a,ą@ 1!", "\\p{N}")
## [[1]]
## [1] "1"
```

1.3.2 Binary properties

Code points may also exhibit a set of Binary Properties. These include:

- ALPHABETIC – alphabetic character;
- ASCII_HEX_DIGIT – a character matching [0-9A-Fa-f];
- LOWERCASE ;
- UPPERCASE ;
- WHITE_SPACE – a space character or TAB or CR or LF or ZWSP or ZWNBSPP; this is not the same as General Category Z.

Note that a code point has only one General Category but may exhibit many binary properties.

```
stri_extract_all_charclass(" \t\n\r", "\\p{Z}")
## [[1]]
## [1] " "
stri_extract_all_charclass(" \t\n\r", "\\p{C}")
## [[1]]
## [1] "\t\n\r"
stri_extract_all_charclass(" \t\n\r", "\\p{WHITE_SPACE}")
## [[1]]
## [1] " \t\n\r"
```

1.4 Summary

There are many character encodings. UTF-8 is the most popular, but you will also encounter other ones from time to time. Fortunately, most of the encodings base on ASCII.

1.5 Bibliography

- ?"stringi-encoding", ?"stringi-search-charclass" + references
- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 9
- Unicode Technical Report #17: *Unicode Character Encoding Model*

Table 5: General Category Properties

Abbr	Long	Description
Lu	Uppercase_Letter	an uppercase letter
Ll	Lowercase_Letter	a lowercase letter
Lt	Titlecase_Letter	a digraphic character, with first part uppercase
LC	Cased_Letter	Lu Ll Lt
Lm	Modifier_Letter	a modifier letter
Lo	Other_Letter	other letters, including syllables and ideographs
L	Letter	Lu Ll Lt Lm Lo
Mn	Nonspace_Mark	a nonspace combining mark (zero advance width)
Mc	Spacing_Mark	a spacing combining mark (positive advance width)
Me	Enclosing_Mark	an enclosing combining mark
M	Mark	Mn Mc Me
Nd	Decimal_Number	a decimal digit
Nl	Letter_Number	a letterlike numeric character
No	Other_Number	a numeric character of other type
N	Number	Nd Nl No
Pc	Connector_Punctuation	a connecting punctuation mark, like a tie
Pd	Dash_Punctuation	a dash or hyphen punctuation mark
Ps	Open_Punctuation	an opening punctuation mark (of a pair)
Pe	Close_Punctuation	a closing punctuation mark (of a pair)
Pi	Initial_Punctuation	an initial quotation mark
Pf	Final_Punctuation	a final quotation mark
Po	Other_Punctuation	a punctuation mark of other type
P	Punctuation	Pc Pd Ps Pe Pi Pf Po
Sm	Math_Symbol	a symbol of mathematical use
Sc	Currency_Symbol	a currency sign
Sk	Modifier_Symbol	a non-letterlike modifier symbol
So	Other_Symbol	a symbol of other type
S	Symbol	Sm Sc Sk So
Zs	Space_Separator	a space character (of various non-zero widths)
Zl	Line_Separator	U+2028 LINE SEPARATOR only
Zp	Paragraph_Separator	U+2029 PARAGRAPH SEPARATOR only
Z	Separator	Zs Zl Zp
Cc	Control	a C0 or C1 control code
Cf	Format	a format control character
Cs	Surrogate	a surrogate code point
Co	Private_Use	a private-use character
Cn	Unassigned	a reserved unassigned code point or a noncharacter
C	Other	Cc Cf Cs Co Cn

- Unicode Standard Annex #44: *Unicode Character Database*
- Unicode Standard Annex #15: *Unicode Normalization Forms*

2 String processing and searching

2.1 Introduction

In this section, we will discuss text processing issues. Why string operations are so important? It turns out, that very often our input data is given in form of text files (e.g. R scripts, C++ code, XML documents, web pages). Also, text is a universal format for data exchange (e.g. between applications). What is more, text data is a source of knowledge (text mining, etc.).

We already know that R character vectors are sequences of strings. Strings are just sequences of bytes. Thus, a character vector is a sequence of sequences of bytes (or integers). Internally, each string ends with byte 0 (such a convention is well-known to C programmers).


```
rawToChar(as.raw(c(65, 66, 67, 0, 68, 69)))
## Error: embedded nul in string: 'ABC\ODE'
```

Note that explicit byte is 0 forbidden.

R maintains a string cache (internally), in which it stores all unique strings (a hash table is used for this purpose). This allows for very fast string comparing, e.g. with “==” or `match()`. Moreover, this reduces the memory consumption.

```
object.size(rep("abcdefg", 1000000))
## 8000088 bytes
object.size(rep("abcdefghijklmno", 1000000))
## 8000096 bytes
```

Note that there are only 8 more bytes, not 8×1000000 more.

 **Exercise:** Write a function `hash()` to generate a hash code of a given string `x`. Use the following algorithm:

```
hash = 0
for each y: a code point in x
    hash = (hash * c + numeric_code(y)) modulo m
return hash as result
```

where $0 < c < m$ – additional arguments to the function, by default equal to $c = 31, m = 2014$. Hash codes may be used for fast string searching (**why?**) – the set of all strings is mapped to $\{0, 1, \dots, m - 1\}$. Generate a set of **many** random words. Choose the **best** c, m empirically.

What about **character encoding issues**? Each string in R has a marked encoding:

- ASCII
- UTF-8
- bytes
- native

Thus, R allows for a peaceful coexistence of strings in different encodings. The “bytes” encoding means no specific encoding, a string should be treated as a sequence of bytes. This is rarely used

“Native” encoding is your system’s default one. By assumption, each string you input via keyboard uses the native encoding. To read the native encoding we type:

```
Sys.getlocale("LC_CTYPE")
## [1] "pl_PL.UTF-8"
```

So on my computer “Native” is the same as “UTF-8”.

```
Encoding("ąść") # unknown == native
## [1] "unknown"
```

Generally, string processing in R is sometimes difficult:

- some functions are not properly vectorized:

```
grep1(c("[a-z]+", "[0-9]+"), c("abc", "123"))
```

```
## Warning: argument 'pattern' has length > 1 and only the first element will be used
## [1] TRUE FALSE
```

- other do not handle NAs properly:

```
nchar(c("ala", "ošć", "", NA)) # number of characters
## [1] 3 3 0 2
paste(letters[1:3], NA) # concatenation
## [1] "a NA" "b NA" "c NA"
```

- some functions return results that are hard to handle:

```
regexec("([a-z]+)=([a-z]+)", "abc obiad=lody def")[[1]]
## [1] 5 5 11
## attr("match.length")
## [1] 10 5 4
```

Hadley Wickam's `stringr` package (quite popular) addresses the above issues. However, base R (and base R-based `stringr`) functions do not enable us to write **portable code** (the same results on different platforms).

E.g. in Slovak, "hladný" < "chladný".

```
oldlocale <- Sys.getlocale("LC_COLLATE")
invisible(Sys.setlocale("LC_COLLATE", "sk_SK.UTF-8"))
"hladny" < "chladny"
## [1] NA
invisible(Sys.setlocale("LC_COLLATE", oldlocale))
```

- The above does not work on Windows, which has different locale specifiers. POSIX use BCP 47 (e.g. "pl_PL.UTF-8" – ISO 639 & ISO 3166-1 alpha-2), and Windows, on the other hand, LCID (e.g. "Polish_Poland.1250") and RFC4646 (\geq Win Vista, np. "en-US").
- A user may not have the Slovak locale installed.

Moreover, base R string processing functions do not necessarily conform to the Unicode standard.

```
toupper("fußball") # FUßBALL?
## [1] "FUßBALL"
"\u0105" == "a\u0328" # Where's Unicode normalization?
## [1] FALSE
```

Because of the above, we have developed the **stringi** package for R.

```
library("stringi")
```

`stringi` heavily relies on the widely-recognized ICU (International Components for Unicode) library.

Before using this package make sure your native encoding and locale have been properly deduced from the R environment's settings:

```
stri_install_check()
## pl_PL.UTF-8; ICU4C 50.1.2; Unicode 6.2
## All tests completed successfully.
```

Read more: `help(package="stringi")`

2.2 Basic string processing tasks

The most basic string processing functions are locale-independent.

String concatenation:

```
stri_paste("advanced", "R")
## [1] "advancedR"
stri_paste("advanced", "R", sep = " ")
## [1] "advanced R"
```

```
stri_paste(c("advanced", "R"), collapse = "")
## [1] "advancedR"
stri_flatten(c("advanced", "R"))
## [1] "advancedR"
stri_paste(c("a", "b"), 1:2, sep = ",", collapse = ";")
## [1] "a,1;b,2"
"a" %stri+% "b"
## [1] "ab"
```

String duplication:

```
stri_dup("a", 1:5)
## [1] "a"      "aa"      "aaa"      "aaaa"      "aaaaa"
```

Substring generation/modification:


```
x <- "abc123aęś"
stri_sub(x, 1, 3)
## [1] "abc"
stri_sub(x, -3)
## [1] "aęś"
stri_sub(x, -3, -2)
## [1] "aę"
stri_sub(x, -3, length=1)
## [1] "a"
stri_sub(x, -3) <- "AES"; x
## [1] "abc123AES"
```


String trimming and padding:

```
stri_trim_both("\t abc\n ")
## [1] "abc"
cat(stri_pad_both(c("abc", "defghij"), 20), sep = "\n")
##      abc
##      defghij
```

Random string generation:

```
stri_rand_shuffle("abcdefghi")
## [1] "fhdbgciea"
stri_rand_strings(3, 5, "[a-z]")
## [1] "olylr" "ocxgb" "iyxsq"
```

 **Exercise:** Write a function `genpwd()` to generate `n` (1 by default) random passwords, each of length `k` (by default 8). Each password should consist of at least 1 lower-case letter, 1 upper-case letter, 1 digit, and 1 code point from an additional set `a` (by default `c("_", "-")`). Do not use `stri_rand*()` functions.

 **Exercise:** Write a function to generate one paragraph of a random “lorem ipsum” text.

The `sprintf()` function:

```
sprintf("Test #%d of %d", 1:3, 3)
## [1] "Test #1 of 3" "Test #2 of 3" "Test #3 of 3"
sprintf("%-20s%.4f", "data science", pi)
## [1] "data science      3.1416"
sprintf("SELECT * FROM %s WHERE id IN (%s)",
  "myTable", stri_paste(sample(1:9, 3), collapse=", "))
## [1] "SELECT * FROM myTable WHERE id IN (9, 6, 5)"
```

Read more: `?sprintf`. (C programmers should know it already)

2.3 Transliteration

More elaborate string processing algorithms are based on an extensive character and rules database.

String transliteration:


```
stri_trans_general("ß", "Name")
## [1] "\\N{LATIN SMALL LETTER SHARP S}"
stri_trans_general("groß@ zółc", "Latin-ASCII")
## [1] "gross(C) zolc"
```

2.4 Locale-dependent operations

What is more, the behavior of some functions takes locale-specific data into account:

String comparing:

```
stri_cmp_lt("hładny", "chładny")
## [1] FALSE
stri_cmp_lt("hładny", "chładny", list(locale="sk_SK"))
## [1] TRUE
stri_cmp_eq("\u0105", "a\u0328") # normalization
## [1] FALSE
stri_sort(c("a", "b", "w", "z"))
## [1] "a" "b" "w" "z"
stri_sort(c("a", "b", "w", "z"), opts=list(locale="et_EE"))
## [1] "a" "b" "z" "w"
```

 **Exercise:** Write a function `numspellout()`, which spells out each number in a given integer vector `x` in your native language. You may assume that each number in `x` is less than 1,000,000,000.

An example for the Polish language:

```
numspellout(c(9:12, 1234))
## [1] "dziewięć"
## [2] "dziesięć"
## [3] "jedenaście"
## [4] "dwanaście"
## [5] "jeden tysiąc dwieście trzydzieści cztery"
```


Text boundary analysis:

```
stri_extract_words("above-mentioned advanced, data; analysis!")
## [[1]]
## [1] "above"      "mentioned" "advanced"  "data"      "analysis"
stri_split_boundaries("above-mentioned advanced, data; analysis!")
## [[1]]
## [1] "above-"      "mentioned " "advanced, " "data; "      "analysis!"
```

See also: `?stri_wrap`.

Case mapping:

```
stri_trans_toupper("groß")
## [1] "GROSS"
stri_trans_totitle("have a nice day")
## [1] "Have A Nice Day"
```

 **Exercise:** You are given a character vector, each string represent a paragraph of text. Write a function `wrap_greedy()`, which implements a greedy word wrap algorithm. Output a character vector such that, when printed on string, it fills up to `h` (76 by default) text columns.

```

text <- c("Tatoooooooo nieeeeeee wraaaaaaaca; ranki i wieeeeeeeeczory
We 1zaaaaaaach goooooooooo czeeeeeeeeeeeeeeeekam",
stri_dup("a ", 70))
h <- 55
cat(wrap_greedy(text, h),
    sep=str_join("\n|", str_dup("-",h-2), "|\\n"))
## Tatoooooooo nieeeeeee wraaaaaaaca; ranki i
## wieeeeeeeeczory We 1zaaaaaaach goooooooooo
## czeeeeeeeeeeeeeeeekam
## |-----|
## a a a a a a a a a a a a a a a a a a a a a a a a a a a a
## a a a a a a a a a a a a a a a a a a a a a a a a a a a a
## a a a a a a a a a a a a a a a a a a a a a a a a a a a a

```

2.5 String searching

stringi currently gives access to **4 search engines**:

- **regex** – regular expressions (t.b.d. later)
- **fixed** – very fast, locale-independent exact substring searching
- **coll** – Collator-based, locale-dependent pattern searching (for natural language processing, slow)
- **charclass** – search for code points from a specific character class (Binary properties, General categories, etc.)

Basic search operations:

- **detect**
- **count**
- **extract_all**, **extract_first**, **extract_last**
- **locate_all**, **locate_first**, **locate_last**
- **replace_all**, **replace_first**, **replace_last**
- **split**

stri_detect_* check if a pattern occurs in a given string:

```

stri_detect_fixed("science", "en")
## [1] TRUE
stri_detect_fixed(c("data", "science"), "en")
## [1] FALSE TRUE
stri_detect_fixed("science", c("en", "em", NA))
## [1] TRUE FALSE NA
stri_detect_fixed(c("data", "science"), c("at", "en"))
## [1] TRUE TRUE

```

Note the difference between the fixed and the coll engine:

```

stri_detect_fixed("a\u0328", "\u0105")
## [1] FALSE
stri_detect_coll("a\u0328", "\u0105")
## [1] TRUE
stri_detect_coll(c("GROSS", "groß"), "gross")
## [1] FALSE FALSE
stri_detect_coll(c("GROSS", "groß"), "gross", list(strength=2))
## [1] TRUE FALSE
stri_detect_coll(c("GROSS", "groß"), "gross", list(strength=1))
## [1] TRUE TRUE

```

stri_count_* count how many times a pattern occurs in a string:

```
stri_count_fixed("a1a2a3", "a")
## [1] 3
stri_count_charclass("a1b2c3d4", "[abc]") # a, b, or c
## [1] 3
stri_count_charclass("a1b2c3d4", "[a-z]") # ASCII range a-z ...
## [1] 4
stri_count_charclass("a1b2ß3ą4", "[a-z]") # ... codes 97-122
## [1] 2
stri_count_charclass("a1b2ß3ą4", "\\p{L}") # Unicode letters
## [1] 4
stri_count_charclass("a1b2ß3ą4", "[^\\p{L}]", "# non-letters
## [1] 4
stri_count_charclass("a1b2ß3ą4", "[^a-z]", "# except 97-122
## [1] 6
```

Read more: ?"stringi-search-charclass".

`stri_extract_*` extract a given pattern match (which is not really interesting for fixed):

```
stri_extract_all_charclass(c("123", "abc", "ąęś123abc"), "\\p{L}")
## [[1]]
## [1] NA
##
## [[2]]
## [1] "abc"
##
## [[3]]
## [1] "ąęś" "abc"
stri_extract_all_charclass("ąęś123abc", "\\p{L}", merge = FALSE)
## [[1]]
## [1] "ą" "ę" "ś" "a" "b" "c"
stri_extract_first_charclass(c("123", "abc", "12ąęś45"), "\\p{L}")
## [1] NA "a" "ą"
```

`stri_locate_*` locate a given pattern:


```
stri_locate_all_fixed(c("abababa", "ba"), "aba")
## [[1]]
##      start end
## [1,]      1  3
## [2,]      5  7
##
## [[2]]
##      start end
## [1,]      NA NA
stri_locate_last_fixed(c("abababa", "ba"), "aba")
##      start end
## [1,]      5  7
## [2,]      NA NA
```

`stri_replace_*` substitute a pattern's occurrence with another string:

```
stri_replace_all_fixed("Python programming", "Python", "R")
## [1] "R programming"
stri_replace_all_fixed("Python", "Python", c("R", "C++"))
## [1] "R" "C++"
stri_replace_all_fixed(c("Python", "Ruby"), c("Python", "Ruby"), c("R", "C++"))
## [1] "R" "C++"
stri_replace_all_charclass("data science ", "\\p{Z}", "")
## [1] "datascience"
```

`stri_split_*` split a string at a pattern's occurrence:

```
stri_split_charclass("data science", "\\p{Z}")
## [[1]]
## [1] "data"      "science"
stri_split_fixed("beware! dogs?", c("!", "!", "e"))
## [[1]]
## [1] "beware" " dogs?"
##
## [[2]]
## [1] "beware" "dogs?"
##
## [[3]]
## [1] "b"      "war"    "! dogs?"
```

 **Exercise:** Write a simple interpreter of expressions written in the Reverse Polish Notation. Given a character vector, a function `rpn()` outputs a numeric vector with the results (of the same length as the input one). Assume that each string consists only of integers and binary operators: `+`, `-`, `*`, `/`, and that each token is separated by a whitespace.

For example:

- `"2 3 +"` gives 5,
- `"3 2 - 1 + 7 *"` gives $((3 - 2) + 1) * 7 = 14$, and
- `"2 7 + 3 / 14 3 - 4 * + 2 /"` results in $((2 + 7)/3 + (14 - 3) * 4)/2 = 23.5$.

Exact string matching:

```
choices <- c("fish", "burrito", "marmalade", "broccoli")
match(c("burrito", "bur", "fish"), choices)
## [1] 2 NA 1
c("burrito", "bur", "fish") %in% choices
## [1] TRUE FALSE TRUE
```

These are quite fast operations, as R's string cache (hash table) is used.

Partial string matching:

```
choices <- c("fish", "burrito", "marmalade", "broccoli")
pmatch("burrito", choices)
## [1] 2
pmatch("bur", choices)
## [1] 2
pmatch("b", choices) # ambiguous
## [1] NA
```

By the way, the `match.arg()` function:

```
test <- function(select = c("two.sided", "less", "greater")) {
  select <- match.arg(select) # explained later
  cat(select)
}


test()
## two.sided
test("less")
## less
test("g")
## greater
test("ggher")
## Error: 'arg' should be one of "two.sided", "less", "greater"
```

Fuzzy matching and the Levenshtein distance:


```
s <- c("niezółty", "niezółtej", "żółtego", "zazółcił",
      "zaczerwienił", "żyły", "żółty", "żółtemu", "zazółconemu")
s[agrep("żółty", s)]
## [1] "niezółty" "niezółtej" "żółtego" "żółty" "żółtemu"

dst <- as.numeric(adist(s, "żółty"))/stri_length("żółty")
names(dst) <- s
dst
##      niezółty      niezółtej      żółtego      zazółcił zaczerwienił
##          0.6          1.0          0.6          1.2          2.4
##          żyły        żółty        żółtemu      zazółconemu
##          0.4          0.0          0.6          1.6
```

See also: the stringdist package.

 **Exercise:** Write a function to “censor” each “bad word” (construct the list yourself) in a given piece of text. Each of the letters, except for first and last one, in a bad word, should be asterisked. For example, if “sheet” and “duck” are bad words, a text “This duck is drawn on a piece of sheet” should be transformed to “This d**k is drawn on a piece of s***t”.

2.6 Summary

Basically, string processing is about working with integer sequences and utilizing a rulebase. Some rules are very complex; luckily, we have ICU and other libraries.

2.7 Bibliography

- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 9
- Unicode Technical Standard #10: *Unicode Collation Algorithm*

3 Regular expressions

3.1 Introduction

Regular expressions are an **extremely powerful** tool for manipulating text data. Here are some examples of what we can easily do with a regex:

- Find a fixed pattern at the beginning or end of a string
- Validate if a given string consists of a decimal number
- Extract all emails or URLs from a piece of text
- Parse "Spec: color=gray, engine=V6" to obtain

```
##      key value
## 1  color  gray
## 2 engine   V6
```

Think for a while how to do the above without regexes.

3.2 Matching single characters

All characters, except:

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

are matched as-is.

```
library("stringi")
stri_count_regex("abc123aAa", c("a", "12"))
## [1] 3 1
```

```
stri_count_regex("abc123aAa", c("a", "12"),
  list(case_insensitive=TRUE))
## [1] 4 1
```

Thus, we have a fixed-pattern search above. If we would like to include a special character, i.e. one of: `.` `\` `|` `(` `)` `[` `{` `}` `^` `$` `*` `+` `?`, in the search pattern, we must escape it with a backslash.

Note:

```
c("\n", "\\n")
## [1] "\n" "\\n"
cat(c("\n", "\\n"))
##
## \n
```

`"\n"` denotes a newline character and `"\\n"` denotes a backslash and `n` (two characters).

For example, to match a dot, we use:

```
stri_count_regex("data...", "\\.")
## [1] 3
```

3.3 Matching single characters

The `"."` metacharacter matches any character

```
stri_count_regex(c("data...", ""), ".")
## [1] 7 0
```

...except for newline (by default).

```
stri_count_regex("data\\nscience", ".")
## [1] 11
stri_count_regex("data\\nscience", ".", list(dotall = TRUE))
## [1] 12
```

This is mostly for historical reasons.

We can also match single characters from a given character set:

```
stri_extract_all_regex("abdwzAWZ12! @", "[adz]")
## [[1]]
## [1] "a" "d" "z"
stri_extract_all_regex("abdwzAWZ12! @", "[\\p{Ll}]") # or \\p{Ll}
## [[1]]
## [1] "a" "b" "d" "w" "z"
stri_extract_all_regex("abdwzAWZ12! @", "[^\\p{Ll}]") # or \\P{Ll}
## [[1]]
## [1] "A" "W" "Z" "1" "2" "!" " " "@"
stri_extract_all_regex("abdwzAWZ12! @", "[\\p{Ll}\\p{Lu}]")
## [[1]]
## [1] "a" "b" "d" "w" "z" "A" "W" "Z"
stri_extract_all_regex("abdwzAWZ12! @", "\\p{Ll}\\p{Lu}")
## [[1]]
## [1] "zA"
```

This works much like in case of `*_charclass` functions.

3.4 Alternation

`"|"` is the **alternation operator**. Its order of precedence is very low.

```
stri_extract_all_regex("abdacaddadab", "ab|dd")
## [[1]]
## [1] "ab" "dd" "ab"
```

Subexpressions may be **grouped** by using parentheses:

```
stri_extract_all_regex("abdacaddadab", "a(b|d)d")
## [[1]]
## [1] "abd" "add"
```

By the way, “(?. . .)” forms a free-format comment:

```
stri_extract_all_regex("abdacaddadab",
  "a(?. match 'a')" %stri+%
  "(b|d)(?. match 'b' or 'd')" %stri+%
  "d(?. match 'a')")
## [[1]]
## [1] "abd" "add"
```

Moreover, “(?i)” turns on a case-insensitive matching:

```
stri_extract_all_regex("abaABaBA", "(?i)ba")
## [[1]]
## [1] "ba" "Ba" "BA"
```

There are more flags available, see the ICU user guide on Regular Expressions.

3.5 Quantifiers

Quantifiers allow us to repeat a subpattern a number of times.

- * – 0 or more times
- + – 1 or more times
- ? – 0 or 1 times
- {n,} – at least n times
- {,m} – at most m times
- {n,m} – at least n but no more than m times

For example, `dog(s?|e|gy)` matches:

- dog
- doge
- dogs
- doggy

Note that alternatives are matched from left-to-right, first match is accepted and there are no retries:


```
stri_extract_first_regex(c("dogs", "dogsy"), "dog(s|sy)")
## [1] "dogs" "dogs"
stri_extract_first_regex(c("dogs", "dogsy"), "dog(sy|s)")
## [1] "dogs" "dogsy"
```

Moreover:

```
stri_extract_all_regex("abaaaabaaabb", "ba*")
## [[1]]
## [1] "baaaa" "baaa" "b" "b"
stri_extract_all_regex("abaaaabaaabb", "ba+")
## [[1]]
## [1] "baaaa" "baaa"
```

Note that a quantifier is applied to a subexpression that precedes it directly (here: a). Subexpressions may be grouped with parentheses:

```
stri_extract_all_regex("aaabababaaaa", "(ab)+")
## [[1]]
## [1] "ababab"
stri_extract_all_regex("aaababbbbabaaaa", "(ab)+")
## [[1]]
## [1] "ababbbbab"
```

 **Exercise:** `?make.names`: A syntactically valid name consists of letters, numbers and the dot or underline characters and starts with a letter or the dot not followed by a number.

Here is a solution:


```
stri_extract_all_regex(c("stri_paste; .abc; de214; pi; .; ..", ".2way; 235"),
  "??? any ideas ???")
```

The above-mentioned quantifiers are **greedy**. In other words, they try to match **as many times as possible**:

```
stri_extract_all_regex("aa(bbb)(ccc)(ddd)eee", "\\(\\.+\\)")
## [[1]]
## [1] "(bbb)(ccc)(ddd)"
```

To match **as few times as possible** (non-greedily), put “?” right after the quantifier:

```
stri_extract_all_regex("aa(bbb)(ccc)(ddd)eee", "\\(\\.+?\\)")
## [[1]]
## [1] "(bbb)" "(ccc)" "(ddd)"
```

 **Exercise:** Match a floating-point number.

```
stri_extract_all_regex("12.123, -53, +1e-9, -1.2423e10, .2, 4.", "??? any ideas ???")
```

3.6 Capturing groups and backreferences


“(...)” forms a **capturing group**. Matches to capturing groups may be extracted with the `stri_match*_regex()` functions:

```
stri_match_all_regex("engine=V6, color=red", "\\(\\w+)=\\(\\w+)")
## [[1]]
##      [,1]      [,2]      [,3]
## [1,] "engine=V6" "engine" "V6"
## [2,] "color=red" "color"  "red"
```

Here `\\w` denotes a *word character*, i.e.:

`[\\{Alphabetic\\}\\p{Mark}\\p{Decimal_Number}\\p{Connector_Punctuation}\\u200c\\u200d]`.

The 1st column is the whole match, the 2nd column – a match from 1st capturing group, and the 3rd column – a match from 2nd capturing group, etc.

 **Exercise:** Extract all text paragraphs from a HTML document.

```
...
<p>This is paragraph 1.</p>
...
<p>This is
paragraph 2.</p>
...
```

“(?:...)” forms a **non-capturing group**.

```
stri_match_all_regex("engine=V6, color=red", "(?:\\w+)=\\(\\w+)")
## [[1]]
##      [,1]      [,2]
## [1,] "engine=V6" "V6"
## [2,] "color=red" "red"
```

By the way, there are no named capturing groups in ICU Regex.

Backreferences may be used to match the same text again. “`\\1`” denotes 1st capturing group match, “`\\2`” – 2nd, etc.

```
stri_extract_all_regex(
  "<strong><em>aaa</em></strong><code>bbb</code><b>eee</a>",
```

```
"\\<([a-z+)]\\>.??\\</\\1\\>")
## [[1]]
## [1] "<strong><em>aaa</em></strong>" "<code>bbb</code>"
```


Also capture groups may be referred to when replacing a substring. In a replacement string, “\$0” denotes the whole match, “\$1” – 1st capturing group match, etc.

```
stri_replace_all_regex(
  "aba abb bab",
  "([a-z])[a-z]\\1",
  "$0@$0")
## [1] "*aba@aba* abb *bab@bab*"
stri_replace_all_regex(
  "<strong><em>aaa</em></strong><code>bbb</code><b>eee</a>",
  "\\<([a-z+)]\\>(.*?)\\</\\1\\>",
  "$2"
)
## [1] "<em>aaa</em>bbb<b>eee</a>"
```

3.7 Anchors, look-aheads, and look-behinds


“^” matches at the start of the string and “\$” matches at the end of the string.

```
stri_detect_regex(c("aaaaab", "baaaab"), "^a+b")
## [1] TRUE FALSE
stri_detect_regex(c("abaaaaab", "baaaab"), "^ba+b$")
## [1] FALSE TRUE
```

 **Exercise:** Test if a given string consists of a valid used name, i.e. if it starts with an ASCII letter, only ASCII letters, digits, or underscores allowed, from 4 to 16 characters.

A solution:

```
stri_detect_regex(c("helenal23", "agh", "_jose_"),
  "^([a-zA-Z][a-zA-Z0-9_]){3,15}$")
## [1] TRUE FALSE FALSE
```

 **Exercise:** Given a list of file paths, omit all but .txt files.


A solution:

```
files <- ...
files[stri_detect_regex(files, "\\..txt$")]
```

“\b” matches at a word boundary.

```
stri_extract_all_regex("123ab643 daf eee!,ad,re4",
  "\\b[a-z]{2,}\\b")
## [[1]]
## [1] "daf" "eee" "ad"
```

Boundaries occur at the transitions between word (\w) and non-word (\W) characters. On the other hand, “\B” matches if the current position is not a word boundary.

 **Exercise:** Trim whitespaces from the beginning and end of a string.

A solution:

```
stri_replace_first_regex(" test ",
  "\\p{White_space}*(.*?)\\p{White_space}*$", "$1")
## [1] "test"
```

 **Exercise:** Match an IP address.

A solution:

```
stri_extract_all_regex("The address is: 127.0.0.1",
  "\\b\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}\\b")
## [[1]]
## [1] "127.0.0.1"
```

Unfortunately, this will also match 999.999.999.999. Do you have any ideas to correct that?

From ICU User Guide on Regexes:

- **(?=...)** – **Look-ahead assertion**
True if the parenthesized pattern matches at the current input position, but does not advance the input position.
- **(?!...)** – **Negative look-ahead assertion**
True if the parenthesized pattern does not match at the current input position. Does not advance the input position.

In other words, look-aheads may be used to ensure that a pattern to match must (must not) be followed by another pattern.

```
stri_extract_all_regex("aba> bab] cde", "[a-z]{3}(?=\\])")
## [[1]]
## [1] "bab"
```

Note that the contents of a look-ahead subpattern are not returned in the result.

From ICU User Guide on Regexes:

- **(?<=...)** **Look-behind assertion**
True if the parenthesized pattern matches text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators.)
- **(?<!=...)** **Negative Look-behind assertion**
True if the parenthesized pattern does not match text preceding the current input position, with the last character of the match being the input character just before the current position. Does not alter the input position. The length of possible strings matched by the look-behind pattern must not be unbounded (no * or + operators.)

In other words, look-behinds may be used to ensure that a pattern to match must (must not) be preceded by another pattern.

```
stri_extract_all_regex("color=gray, gearbox is auto",
  "(?<=) [a-z]+")
## [[1]]
## [1] "gray"
stri_extract_all_regex("[aba] <dad> [euf> def",
  "(?<=) [a-z]{3}")
## [[1]]
## [1] NA
```

Exercise: Match an email.

A solution:

```
stri_detect_regex(c("gagolews@rexamine.com", "me@localhost"),
  "~([a-z0-9_.-]+)@([\\da-z.-]+)\\.([a-z.]{2,6})$")
## [1] TRUE FALSE
```

Exercise: Extract an URL.

A solution:

```
stri_extract_all_regex("for more info, visit: www.rexamine.com!",  
  "(https?:/)?(\\da-z.-+)?\\.([a-z.]{2,6})([/\\w \\.-]*)*/?"  
## [[1]]  
## [1] "www.rexamine.com"
```

3.8 Summary

Regex have many advantages:

- Most language support regexes,
- Write less, do more,
- May be used directly in most text editors (see `<CTRL+F>` in RStudio).

Please be aware of regex limitations. First of all a regex engine is not a full-fledged parser. You cannot parse R or XML code with a regex. Moreover, regexes may be used for matching the so-called *regular languages*.

Also keep in mind that Regex engines differ from each other. R is shipped with two:

- TRE (ERE-like, default),
- PCRE (Perl-Compatible Regex).

R base functions: `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, `sub()`, `gsub()`, `gregexpr()`.

On the other hand, `stringi` uses the ICU regex (Java-like) engine. Even though simple regexes are engine-independent, there are not fully compatible with each other.

3.9 Bibliography

- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 9
- Hopcroft J.E., Motwani R., Ullman J.D., *Introduction to Automata Theory, Languages, and Computation*, Prentice Hall, 2006
- *Regular Expressions – ICU User Guide*, userguide.icu-project.org/strings/regexp
- Friedl J.E.F., *Mastering Regular Expressions*, O'Reilly Media, 2006
- Unicode Technical Standard #18: *Unicode Regular Expressions*
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 11

4 Date and time

4.1 Introduction

There are 3 compound types to represent date and/or time:

- Date
- POSIXct
- POSIXlt


None of them is built upon strings, but our daily data science activities often consist of date/time:

- parsing (from strings),
- formatting (to strings).

The `Sys.Date()` function returns the current date. The `Sys.time()` function returns the current date and time.

```
(sysDate <- Sys.Date())  
## [1] "2014-08-10"  
(systime <- Sys.time())  
## [1] "2014-08-10 22:37:27 CEST"
```

ISO 8601 defines a 24-hour format of the form YYYY-mm-dd HH:MM:SS. Note how time zones are denoted: e.g. UTC is the Universal Time Coordinated, CET is the Central European Time, and CEST is the Central European Summer Time.

 **Exercise:** Write a regex to match a ISO 8601 (YYYY-mm-dd HH:MM:SS) formatted date/time. Try not to match as many types of incorrect dates as possible. What are the limitations of regexes here?

4.2 Dealing with dates

The `sysDate` object is in fact...

```
class(sysDate)
## [1] "Date"
typeof(sysDate)
## [1] "double"
unclass(sysDate)
## [1] 16292
```

This is the number of days since the UNIX Epoch (1970-01-01 00:00:00 UTC).

The `as.Date()` function may be used to convert a string to a Date object:

```
unclass(as.Date("1970-01-01"))
## [1] 0
unclass(as.Date("1939-09-01"))
## [1] -11080
unclass(as.Date("1970/12/31"))
## [1] 364
unclass(as.Date("2014"))
## Error: character string is not in a standard unambiguous format
unclass(as.Date("2012-02-29"))
## [1] 15399
unclass(as.Date("2011-02-29"))
## Error: character string is not in a standard unambiguous format
```

4.3 Dealing with date/time objects

Now let us take a deeper look at the `systemtime` object:

```
class(systemtime)
## [1] "POSIXct" "POSIXt"
typeof(systemtime)
## [1] "double"
unclass(systemtime)
## [1] 1407703047
print(unclass(systemtime), digits = 22)
## [1] 1407703047.030994176865
```

Now we have the number of second since the UNIX Epoch. `POSIXct` means POSIX calendar time.

A simple conversion:

```
as.POSIXct("2014-12-31 23:59:59")
## [1] "2014-12-31 23:59:59 CET"
as.POSIXct("2014-12-31")
## [1] "2014-12-31 CET"
as.POSIXct("1970-01-01 00:00:00")
## [1] "1970-01-01 CET"
unclass(as.POSIXct("1970-01-01 00:00:00"))
## [1] -3600
```



```
## attr(,"tzone")
## [1] ""
structure(1e-05, class = c("POSIXct", "POSIXt"), tzone = "UTC")
## [1] "1970-01-01 00:00:00 UTC"
```

Note that our local time zone is used while parsing dates.

By the way, `POSIXct` and `Date` are atomic vectors. This means that they may be included in a data frame:

```
data.frame(id=c("a", "b"),
  d=as.Date(c("1915-12-31", "1959-04-23")),
  t=as.POSIXct(c("2013-01-01 12:42:21", "2014-05-12 11:29:14")))
##      id      d      t
## 1    a 1915-12-31 2013-01-01 12:42:21
## 2    b 1959-04-23 2014-05-12 11:29:14
```

Another date/time representation is provided by `POSIXlt` (POSIX local time) objects:

```
as.POSIXlt(systime)
## [1] "2014-08-10 22:37:27 CEST"
class(as.POSIXlt(systime))
## [1] "POSIXlt" "POSIXt"
typeof(as.POSIXlt(systime))
## [1] "list"
str(unclass(as.POSIXlt(systime)))
## List of 11
## $ sec   : num 27
## $ min   : int 37
## $ hour  : int 22
## $ mday  : int 10
## $ mon   : int 7
## $ year  : int 114
## $ wday  : int 0
## $ yday  : int 221
## $ isdst : int 1
## $ zone  : chr "CEST"
## $ gmtoff: int 7200
## - attr(*, "tzone")= chr [1:3] "" "CET" "CEST"
```

Note the `mon` and `yday` fields.

More fancy date/time formatting and parsing is available via `strptime()` and `strptime()`.

```
strptime("1970-01-01 00:00:00", "%Y-%m-%d %H:%M:%S", tz="UTC")
## [1] "1970-01-01 UTC"
unclass(as.POSIXct(
  strptime("1970-01-01 00:00:00", "%Y-%m-%d %H:%M:%S", tz="UTC")))
## [1] 0
## attr(,"tzone")
## [1] "UTC"
strptime("01:43:12", "%H:%M")
## [1] "2014-08-11 01:43:00 CEST"
strptime(systime, "%x %X")
## [1] "10.08.2014 22:37:27"
```

See `?strptime` for a list of format specifiers.


Date formatting may be locale-dependent:

```
oldloc <- Sys.getlocale("LC_TIME")
strptime(systime, "%x %X (%A, %B)")
## [1] "10.08.2014 22:37:27 (niedziela, sierpień)"
```

```
Sys.setlocale("LC_TIME", "en_US.UTF8")
## [1] "en_US.UTF8"
strftime(systime, "%x %X (%A, %B)")
## [1] "08/10/2014 10:37:27 PM (Sunday, August)"
Sys.setlocale("LC_TIME", "de_DE.UTF8")
## [1] "de_DE.UTF8"
strftime(systime, "%x %X (%A, %B)")
## [1] "10.08.2014 22:37:27 (Sonntag, August)"
Sys.setlocale("LC_TIME", oldloc)
## [1] "pl_PL.UTF-8"
```

Moreover:

```
weekdays(systime)
## [1] "niedziela"
months(systime)
## [1] "sierpień"
quarters(systime)
## [1] "Q3"
```

 **Exercise:** Write a function that returns a readable textual representation of today's date in your native language, e.g. "Today is Thursday, January 1, 2015", "Dziś jest czwartek, 1 stycznia 2015", or "Heute ist Donnerstag, der 1. Januar 2015".


To add more fun, try not to use `strftime()`. You will have some intuition on how much hard work has this function to do.

4.4 Date/time arithmetic

Some basic arithmetic operations have been overloaded for date/time objects:

```
sysDate - as.Date("2014-01-01")
## Time difference of 221 days
str(sysDate - as.Date("2014-01-01"))
## Class 'difftime' atomic [1:1] 221
## .. attr(*, "units")= chr "days"
seq(as.Date("2014-01-01"), sysDate, length.out = 4)
## [1] "2014-01-01" "2014-03-15" "2014-05-28" "2014-08-10"
```

See also: `quantile()`, `cut()`, `round()`, `hist()`,

 **Exercise:** You are given a character vector `bday` with birthday dates (yyyy-mm-dd) of a set of people.

Write a function that returns a data frame with `length(bday)` rows and the following 4 columns:

- `year` – integer;
- `month` – integer;
- `day` – integer;
- `adult` – logical; whether a person is today at least 18 years old.

4.5 Summary

You will be dealing with date and time processing in your daily data science activities quite often. Make sure you know how to parse, format, and transform dates, especially English dates in non-English locales.

See also: the `lubridate` and `chron` packages.

4.6 Bibliography

- Gagolewski M., *Programowanie w języku R*, PWN, 2014 (in Polish), Chap. 9