R – Extracting Text

R-Basic String Manipulation

- Basic Manipulation means transforming and processing strings in such way that they do not require the use of regular expressions.
- More advanced manipulations involve defining patterns of text and matching such patterns.
- Besides creating and printing strings, there are a number of very handy functions in R for doing some basic manipulation of strings.

Function	Description
nchar()	number of characters
tolower()	convert to lower case
toupper()	convert to upper case
casefold()	case folding
chartr()	character translation
abbreviate()	abbreviation
substring()	substrings of a character vector
substr()	substrings of a character vector

R- using stringr Package

- Stringr is a set of simple wrappers that make R's string functions more consistent, simpler and easier to use.
- By ensuring function and argument names (and positions) are consistent, all functions deal with NA's and zero length character appropriately, and the output data structures from each function matches the input data structures of other functions.
- stringr provides functions for both basic manipulation functions and functions for regular expression operations

Function	Description	Similar to
str_c()	string concatenation	paste()
str_length()	number of characters	nchar()
str_sub()	extracts substrings	substring()
str_dup()	duplicates characters	none
str_trim()	removes leading and trailing whitespace	none
str_pad()	pads a string	none
str_wrap()	wraps a string paragraph	strwrap()
str_trim()	trims a string	none

R-stringr functions

- Notice that all functions in stringr start with "str_" followed by a term associated to the task they perform.
  - For example, str_length() gives you the number (i.e. length) of characters in a string.
- In addition, some functions are designed to provide a better alternative to already existing functions.
- This is the case of str_length() which is intended to be a substitute of nchar().
- Other functions, don't have a corresponding alternative such as str_dup() which allows you to duplicate characters.

Str_c()

- This function is equivalent to paste() but instead of using the white space as the default separator, str_c() uses the empty string "" which is a more common separator when pasting strings:

```
# default usage
> str_c("May", "The", "Force", "Be", "With", "You")
#> [1] "MayTheForceBeWithYou"
# removing zero length objects
> str_c("May", "The", "Force", NULL, "Be", "With", "You", character(0))
#> [1] "MayTheForceBeWithYou"
```

- Another major difference between str_c() and paste(): zero length arguments like NULL and character(0) are silently removed by str_c().
- We can change the default separator:

```
# changing separator
> str_c("May", "The", "Force", "Be", "With", "You", sep = "_")
#> [1] "May_The_Force_Be_With_You"
```

Str_length()

- Consistent behavior when dealing with NA values

```
# some text (NA included)
> some_text <- c("one", "two", "three", NA, "five")
# compare 'str_length' with 'nchar', both behave as same
> nchar(some_text)
#> [1]  3  3  5 NA  4
> str_length(some_text)
#> [1]  3  3  5 NA  4
```

Str_length()

- In addition, str_length() has the nice feature that it converts factors to characters, something that nchar() is not able to handle:

```
> some_factor <- factor(c(1,1,1,2,2,2), labels = c("good", "bad"))
> print(some_factor)
#> [1] good good good bad  bad  bad
#> Levels: good bad

> nchar(some_factor)# try 'nchar' on a factor
#> Error in nchar(some_factor): 'nchar()' requires a character vector

# now compare it with 'str_length'
> str_length(some_factor)
#> [1] 4 4 4 3 3 3
```

Str_sub()

- To extract substrings from a character vector stringr provides str_sub() which is equivalent to substring(). The function str_sub() has the following usage:

```
str_sub(string, start = 1L, end = -1L)
```

- The three arguments in the function are:
  - a string vector,
  - a start value indicating the position of the first character in substring, and
  - an end value indicating the position of the last character.

- Examples:

```
> lorem <- "Lorem Ipsum"                    # apply 'str_sub'
> str_sub(lorem, start = 1, end = 5)        #> [1] "Lorem"

# equivalent to 'substring'
> substring(lorem, first = 1, last = 5)     #> [1] "Lorem"

# another example
> str_sub("adios", 1:3)            #> [1] "adios" "dios"  "ios"
```

Str_sub()

- It has the ability to work with negative indices in the start and end positions. When use a negative position, str_sub() counts backwards from last character:

```
> resto = c("brasserie", "bistrot", "creperie", "bouchon")

# 'str_sub' with negative positions
> str_sub(resto, start = -4, end = -1)
#> [1] "erie" "trot" "erie" "chon"

# compared to substring (useless)
> substring(resto, first = -4, last = -1)
#> [1] "" "" "" ""
```

Str_sub()

```
# extracting sequentially
> str_sub(lorem, seq_len(nchar(lorem)))
#>  [1] "Lorem Ipsum" "orem Ipsum" "rem Ipsum" "em Ipsum" "m Ipsum"
#>  [6] " Ipsum" "Ipsum" "psum" "sum" "um"
#> [11] "m"

> substring(lorem, seq_len(nchar(lorem)))
#>  [1] "Lorem Ipsum" "orem Ipsum" "rem Ipsum" "em Ipsum" "m Ipsum"
#>  [6] " Ipsum" "Ipsum" "psum" "sum" "um"
#> [11] "m"
```

- It can accept a set of positions which will be recycled over the string. Better way is to give str_sub() a negative sequence:

Str_sub()

```
# reverse substrings with negative positions
> str_sub(lorem, -seq_len(nchar(lorem)))
#> [1] "m" "um" "sum" "psum" "Ipsum"
#> [6] " Ipsum" "m Ipsum" "em Ipsum" "rem Ipsum" "orem Ipsum"
#> [11] "Lorem Ipsum"

> substring(lorem, -seq_len(nchar(lorem)))
#>  [1] "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum"
#>  [6] "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum" "Lorem Ipsum"
#> [11] "Lorem Ipsum"
```

Str_sub()

- We can use str_sub() for replacing substrings:

```
# replacing 'Lorem' with 'Nullam'
> lorem <- "Lorem Ipsum"
> str_sub(lorem, 1, 5) <- "Nullam"
> lorem                     #> [1] "Nullam Ipsum"

# replacing with negative positions
> lorem <- "Lorem Ipsum"
> str_sub(lorem, -1) <- "Nullam"
> lorem                     #> [1] "Lorem IpsuNullam"

> lorem <- "Lorem Ipsum"          # multiple replacements
> str_sub(lorem, c(1,7), c(5,8)) <- c("Nullam", "Enim")
> lorem             #> [1] "Nullam Ipsum"  "Lorem Enimsum"
```

Str_dup()

- A common operation when handling characters is duplication.
- The problem is that R doesn't have a specific function for that purpose.
- str_dup() duplicates and concatenates strings within a character vector.
- Syntax: `str_dup(string, times)`
- Parameters are:
  - The first input is the string that you want to dplicate.
  - The second input, times, is the number of times to duplicate each string:

```
# default usage
> str_dup("hola", 3)              #> [1] "holaholahola"

# use with differetn 'times'
> str_dup("adios", 1:3)
#> [1] "adios"           "adiosadios"      "adiosadiosadios"
```

Str_dup()

```
# use with a string vector
> words <- c("lorem", "ipsum", "dolor", "sit", "amet")

> str_dup(words, 2)
#> [1] "loremlorem" "ipsumipsum" "dolordolor" "sitsit"
"ametamet"


> str_dup(words, 1:5)
#> [1] "lorem" "ipsumipsum" "dolordolordolor"
#> [4] "sitsitsitsit" "ametametametametamet"
```

Str_pad()

- str_pad() is used for padding a string. Its default usage has the following form:

```
> str_pad(string, width, side = "left", pad = " ")
```

- It accepts a string and pad it with leading or trailing characters to a specified total width.
- The default padding character is a space (pad = " "), and consequently the returned string will appear to be either left-aligned (side = "left"), right-aligned (side = "right"), or both (side = "both").

```
# default usage
> str_pad("hola", width = 7)
#> [1] "   hola"
```

Str_pad()

```
# pad both sides
> str_pad("adios", width = 7, side = "both")
#> [1] " adios "

# left padding with '#'
> str_pad("hashtag", width = 8, pad = "#")
#> [1] "#hashtag"

# pad both sides with '-'
> str_pad("hashtag", width = 9, side = "both", pad = "-")
#> [1] "-hashtag-"
```

Str_wrap()

- The function str_wrap() is equivalent to strwrap() which can be used to wrap a string to format paragraphs.
- The idea of wrapping a (long) string is to first split it into paragraphs according to the given width, and then add the specified indentation in each line (first line with indent, following lines with exdent). Its default usage has the following form:

```
> str_wrap(string, width= 80, indent= 0, exdent =0)
```

- For instance, consider the following quote (from Douglas Adams) converted into a paragraph:

```
# quote (by Douglas Adams)
> some_quote<-c( "I may not have gone", "where I intended to
go,", "but I think I have ended up", "where I needed to be")

# some_quote in a single paragraph
> some_quote <- paste(some_quote, collapse = " ")
```

Str_wrap()

- Now, say you want to display the text of some_quote within some pre-specified column width (e.g. width of 30). You can achieve this by applying str_wrap() and setting the argument width = 30

```
# display paragraph with width=30
> cat(str_wrap(some_quote, width = 30))
#> I may not have gone where I
#> intended to go, but I think I
#> have ended up where I needed
#> to be
```

Str_wrap()

- Besides displaying a (long) paragraph into several lines, you may also wish to add some indentation. Here's how you can indent the first line, as well as the following lines:

```
# display paragraph with first line indentation of 2
> cat(str_wrap(some_quote, width = 30, indent = 2), "\n")
#>   I may not have gone where I
#> intended to go, but I think I
#> have ended up where I needed
#> to be

# display paragraph with following lines indentation of 3
> cat(str_wrap(some_quote, width = 30, exdent = 3), "\n")
#> I may not have gone where I
#>    intended to go, but I think I
#>    have ended up where I needed
#>    to be
```

Str_trim()

- One of the typical tasks of string processing is that of parsing a text into individual words. Usually, you end up with words that have blank spaces, called whitespaces, on either end of the word.
- In this situation, you can use the str_trim() function to remove any number of whitespaces at the ends of a string. Its usage requires only two arguments:

```
> str_trim(string, side = "both")
```

- The first input is the string to be strimmed, and the second input indicates the side on which the whitespace will be removed.
- Consider the following vector of strings, some of which have whitespaces either on the left, on the right, or on both sides. Here's what str_trim() would do to them under different settings of side

```
# text with whitespaces
> bad_text <- c("This", " example ", "has several
", "   whitespaces ")
```

Str_trim()

```
# remove whitespaces on the left side
> str_trim(bad_text, side = "left")
#> [1] "This"           "example "       "has several    "
"whitespaces "

# remove whitespaces on the right side
> str_trim(bad_text, side = "right")
#> [1] "This"           " example"       "has several"    "
whitespaces"

# remove whitespaces on both sides
> str_trim(bad_text, side = "both")
#> [1] "This"        "example"     "has several"
"whitespaces"
```

Word extraction with word()

- The word() function that is designed to extract words from a sentence:

```
> word(string, start = 1L, end = start, sep = fixed(" "))
```

- The way in which you use word() is by passing it a string, together with a start position of the first word to extract, and an end position of the last word to extract.
- By default, the separator sep used between words is a single space.

```
# some sentence
change <- c("Be the change", "you want to be")

# extract first word
> word(change, 1)
#> [1] "Be"  "you"
```

Word extraction with word()

```
# extract second word
> word(change, 2)
#> [1] "the"  "want"

# extract last word
> word(change, -1)
#> [1] "change" "be"

# extract all but the first words
> word(change, 2, -1)
#> [1] "the change" "want to be"
```

Other functions

- Str_detect(): Detect the presence or absence of a pattern in a string.
- Str_split():
- Str_trim()
- Str_extract()
- Str_replace()
- Str_replace_all()