ISOM 3390: Business Programming in R

Instructor: Bingjie Qian (PhD candidate)

Summer 2020, HKUST

Topic 9: String Operations

9.1 Strings

Why do we care about strings (or more generally, text)?

- A lot of interesting data out there is in character form!
    - Webpages, emails, surveys, logs, search queries, etc.
- Even if we just care about numbers eventually, we'll need to understand how to get numbers from text.
- Strings do play a big role in many data cleaning and preparation tasks.

What are strings in R?

A string is a sequence of characters bound together.

```
typeof("R")
```

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

```
typeof("Business Programming in R")
```

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

Make Strings

To make strings, just use *double quotes* or *single quotes* and type anything in between.

```
str.1 <- "Business"
str.2 <- 'Programming'
```

Double quotes are usually preferred and character constants are printed using double quotes:

```
str.1
```

```
## [1] "Business"
```

```
str.2
```

```
## [1] "Programming"
```

To see the raw content of the string, use `writeLines()`:

```
writeLines(str.1)
```

```
## Business
```

```
writeLines(str.2)
```

```
## Programming
```

To include a literal single or double quote in a string, we use `\` (backslash) to **escape** it. Single quotes need to be escaped by backslash in single-quoted strings, and double quotes in double-quoted strings.

```
str.3 <- "It's not right."
str.4 <- 'It\'s not right.'
str.5 <- "\"It's not right\", they said."
str.5
```

```
## [1] "\"It's not right\", they said."
```

```
writeLines(str.5)
```

```
## "It's not right", they said.
```

Because `\` is used as the escape, if we want to include a literal backslash, we need to double it up (`\\`).

```
x <- c("\"", "\\")
x
```

```
## [1] "\"" "\\"
```

```
writeLines(x)
```

```
## "
## \
```

Some other escape sequence that starts with backslash include (?'"' or ?"'" to see the complete list):

- newline
- carriage return
- tab

They count as characters and can be included in strings:

```
message <- "Dear Students,\n\nWelcome to ISOM3390!"
writeLines(message)
```

```
## Dear Students,
##
## Welcome to ISOM3390!
```

Coercion

Make things into strings with `as.character()` :

```
as.character(0.8)
```

```
## [1] "0.8"
```

```
as.character(TRUE)
```

```
## [1] "TRUE"
```

```
as.character(1:5)
```

```
## [1] "1" "2" "3" "4" "5"
```

Make a string into other data type (depending on the given string, of course):

```
as.numeric("0.5")
```

```
## [1] 0.5
```

```
as.numeric("Hi!")      # NA
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

```
as.logical("True")
```

```
## [1] TRUE
```

```
as.logical("TRU")      # NA
```

```
## [1] NA
```

```
as.Date("2020/1/22")
```

```
## [1] "2020-01-22"
```

9.2 String Operations with `stringr`

Base R provides a solid set of string operations, but

- They can be inconsistent and a little hard to learn and remember.
- Additionally, they lag behind the string operations in other programming languages (like Python).

The `stringr` package acts as simple wrappers that make R's string functions more consistent, simpler, and easier to use.

- Functions from `stringr` have more intuitive names, and all start with `str_` and take a vector of strings as the first argument.
- It simplifies string operations by eliminating options that we don't need 95% of the time.
- It is a package in the core `tidyverse`, and works well in conjunction with the pipe `%>%`.

```
library("tidyverse")
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
```

```
## ✓ ggplot2 3.3.2     ✓ purrr   0.3.4
## ✓ tibble  3.0.1     ✓ dplyr   1.0.0
## ✓ tidyr   1.1.0     ✓ stringr 1.4.0
## ✓ readr   1.3.1     ✓ forcats 0.5.0
```

```
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

A list of functions that start with `str_`:

```
apropos("str_")
```

```
##  [1] "str_c"           "str_conv"        "str_count"       "str_detect"
##  [5] "str_dup"         "str_ends"        "str_extract"     "str_extract_all"
##  [9] "str_flatten"     "str_glue"        "str_glue_data"   "str_interp"
## [13] "str_length"      "str_locate"      "str_locate_all"  "str_match"
## [17] "str_match_all"   "str_order"       "str_pad"         "str_remove"
## [21] "str_remove_all"  "str_replace"     "str_replace_all" "str_replace_na"
## [25] "str_sort"        "str_split"       "str_split_fixed" "str_squish"
## [29] "str_starts"      "str_sub"         "str_sub<-"       "str_subset"
## [33] "str_to_lower"    "str_to_sentence" "str_to_title"    "str_to_upper"
## [37] "str_trim"        "str_trunc"       "str_view"        "str_view_all"
## [41] "str_which"       "str_wrap"
```

Count the Number of Characters: `str_length()`

Use `str_length()` (or `nchar()` in base R) to count the number of characters in a string:

```
length("code monkey")       # length of the vector
```

```
## [1] 1
```

```
str_length("code monkey")   # or `nchar()` in base R
```

```
## [1] 11
```

```
str_length(c("R","Business Programming", NA))   # it vectorizes
```

```
## [1]  1 20 NA
```

Combine Strings: `str_c()`

Use the `str_c()` function (or `paste()` in base R) to join two (or more) strings into one. Use the `sep` argument to control how they're separated:

```
str_c("Spider", "Man")      # default to having no separator
```

```
## [1] "SpiderMan"
```

```
str_c("Spider", "Man", sep = "-")
```

```
## [1] "Spider-Man"
```

```
str_c("Good ", c("morning", "afternoon", "evening"), "!")   # it vectorizes
```

```
## [1] "Good morning!"   "Good afternoon!" "Good evening!"
```

Condense a vector of strings into one big string by using the `collapse` argument:

```
presidents <- c("Trump", "Obama", "Bush", "Clinton", "Bush", "Reagan", "Carter", "Ford")
str_c(presidents, collapse = "; ")
```

```
## [1] "Trump; Obama; Bush; Clinton; Bush; Reagan; Carter; Ford"
```

Get a Substring: `str_sub()`

Extract parts of a string using `str_sub()` (or `substr()` in base R). It takes `start` and `end` arguments which give the (inclusive) position of the substring.

```
str_sub("Apple", 1, 3)
```

```
## [1] "App"
```

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)     # it vectorizes
```

```
## [1] "App" "Ban" "Pea"
```

```
str_sub(x, -3, -1)   # negative numbers count backwards from right side
```

```
## [1] "ple" "ana" "ear"
```

```
str_sub("R", 1, 5)   # it won't fail if the string is too short
```

```
## [1] "R"
```

Can be used with the assignment operator to modify strings:

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
```

```
## [1] "apple"  "banana" "pear"
```

In the above code, `str_to_lower()` is used to convert a string to lower case. Similar functions include `str_to_upper()`, `str_to_title()`, and `str_to_sentence()`.

An example: Trump's Words

Use `read_lines()` to read the file `trump.txt`, which returns a character vector with one element for each line.

```
trump.lines <- read_lines("trump.txt")
class(trump.lines)    # a character vector
```

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

```
length(trump.lines)
```

```
## [1] 113
```

```
trump.lines[1:5]
```

```
## [1] "Friends, delegates and fellow Americans: I humbly and gratefully accept your nomination for the presidenc
y of the United States."
## [2] "Story Continued Below"
## [3] ""
## [4] "Together, we will lead our party back to the White House, and we will lead our country back to safety, pr
osperity, and peace. We will be a country of generosity and warmth. But we will also be a country of law and orde
r."
## [5] "Our Convention occurs at a moment of crisis for our nation. The attacks on our police, and the terrorism
in our cities, threaten our very way of life. Any politician who does not grasp this danger is not fit to lead ou
r country."
```

Make it one long string:

```
trump.text <- trump.lines %>% str_c(collapse = " ")   # one long string
trump.text %>% str_sub(1, 128)   # the first 128 characters
```

```
## [1] "Friends, delegates and fellow Americans: I humbly and gratefully accept your nomination for the presidenc
y of the United States."
```

Split a String

`str_split()` splits a string into pieces. The input can be a string or a vector of strings. Because each component of the vector might contain a different number of pieces, it returns a list.

```
fruits <- c(
  "apples and oranges and pears and bananas",
  "pineapples and mangos and guavas"
)
str_split(fruits, " and ")   # return a list of two vectors
```

```
## [[1]]
## [1] "apples"  "oranges" "pears"   "bananas"
##
## [[2]]
## [1] "pineapples" "mangos"     "guavas"
```

Split the long string `trump.text` into words (separated by spaces):

```
trump.words <- str_split(trump.text, " ")   # return a list of one vector
str(trump.words)
```

```
## List of 1
##  $ : chr [1:4437] "Friends," "delegates" "and" "fellow" ...
```

```
trump.words <- trump.words[[1]]
str(trump.words)    # a vector
```

```
##  chr [1:4437] "Friends," "delegates" "and" "fellow" "Americans:" "I" ...
```

We can use `table()` to show the word counts, which is the most basic tool for summarizing text.

```
trump.wordtab <- table(trump.words)
trump.wordtab[61:78]
```

```
## trump.words
##          address   Administration  Administration, Administration's
##                2                2                1                1
##            admit          advance         advocate          affairs
##                1                1                1                1
##         affected         afflicts           afford          African
##                1                1                1                1
## African-American            after            After           again,
##                2                5                2                1
##           Again,           again.
##                1                6
```

But, some include punctuation marks, and are not all actual words. We need to better split text with the use of **regular expressions**.

[Task 1: String Operations]

a. Consider the string vector `presidents` of length 5 below, containing the last names of past US presidents. Convert the first letter of each name in `presidents` to upper case.

```
presidents <- c("trump", "obama", "bush", "clinton", "bush")
```

b. Consider the string `presidents` defined below. Use `str_split()` to split this string up into a string vector of length 5, with elements "Trump", "Obama", "Bush", "Clinton", and "Bush". Then, sort the names in alphabetical (increasing) order and use `str_c()` to combine this string vector into a single string "Bush -> Bush -> Clinton -> Obama -> Trump".

```
presidents <- "Trump, Obama, Bush, Clinton, Bush"
```

[End of Task 1]

9.3 Regular Expressions

Pattern Matching

```
Mark,
Good speaking with you. I'll follow up when I get your email.
Thanks,

Rosanna
Rosanna Migliaccio
Vice President
Robert Walters Associates
(212) 704-9900
Fax: (212) 704-4312
mailto:rosanna@robertwalters.com
http://www.robertwalters.com
```

How could we match a phone number? an email address? a URL?

Each of these types of data have a fairly regular pattern that we can easily pick out by eye.

But how can we pick them programmatically?

Regular expressions (**regexps** or **regexs** for short) are a concise language for describing patterns of text.

We can use `str_view()` and `str_view_all()` functions to view regular expression match, which take a character vector and a regular expression, and show us how they match with HTML rendering. `str_view()` shows the first match; `str_view_all()` shows all the matches.

Basic Matching

The simplest patterns match **exact strings**.

```
x <- c("apple", "banana", "pear")
str_view_all(x, "an")
```

apple

ban ana

pear

Matches never overlap. Use `str_view_all` to see how many times will the pattern `"ana"` match in `"banana"`:

```
str_view_all(x, "ana")
```

apple

b ana na

pear

An period/dot `.` matches *any character* except a newline:

```
str_view_all(x, ".a.")
```

apple

ban ana

`pear`

Alternation

`|` is the alternation operator, which will pick between one or more possible matches. For example, `"fly|flies"` will match `"fly"` or `"flies"`.

```
x <- c("time flies", "fruitfly")
str_view(x, "fly|flies")
```

`time flies`

`fruitfly`

Concatenation

Concatenation of regexps is a regexp. E.g., `"(time|fruit)(fly|flies)"` tries to match `"time"` or `"fruit"`, then `"fly"` or `"flies"`. Parentheses define groups (more on this later).

```
str_view(x, "(time|fruit)(fly|flies)")
```

`time flies`

`fruitfly`

```
str_view(x, "(time|fruit) (fly|flies)")
```

`time flies`

`fruitfly`

Character Class

Square braces `[]` are used to define a **character class**, and indicate that we want to match anything inside the square braces for *one character position*.

E.g, `"[AEIOU]"` matches the `"I"` and `"O"` in `"ISOM3390"`; `"[789]"` matches the `"9"` in `"ISOM3390"`.

```
str_view_all(c("ISOM3390", "MARK3220"), "[AEIOU]")
```

`ISOM3390`

`MARK3220`

A dash `-` inside square braces denotes a **range**. E.g., `"[a-e]"` is the same as `"[abcde]"`; `"[0-9]"` is the same as `"[0123456789]"`.

```
str_view_all(c("ISOM3390", "MARK3220"), "[A-Z][0-9]")
```

`ISOM3390`

`MARK3220`

A caret `^` inside square braces **negates** what follows. E.g., `"[^0-9]"` matches anything but a digit.

```
str_view_all(c("ISOM3390","MARK3220"), "[A-Z][^0-9][^0-9][0-9]")
```

`ISOM3390`

`MARK3220`

There are some **predefined character classes** to specify entire classes of characters.

```
* [:digit:] or \d: digits, equivalent to [0-9].
* \D: non-digits, equivalent to [^0-9].
* [:lower:]: lower-case letters, equivalent to [a-z].
* [:upper:]: upper-case letters, equivalent to [A-Z].
* [:alpha:]: alphabetic characters, equivalent to [[:lower:][:upper:]] or [A-z].
* [:alnum:]: alphanumeric characters, equivalent to [[:alpha:][:digit:]] or [A-z0-9].
* \w: word characters, equivalent to [[:alnum:]_].
* \W: not word, equivalent to [^A-z0-9_].
* [:xdigit:]: hexadecimal digits (base 16), equivalent to [0-9A-Fa-f].
* [:blank:]: blank characters, i.e. space and tab.
* [:space:] or \s: whitespace characters (tab, newline, vertical tab, form feed, carriage return, space).
* \S: not whitespaces.
* [:punct:]: punctuation characters (period, question mark, exclamation point, comma, semicolon, colon, dash, hyp
hen, parentheses, brackets, braces, apostrophe, quotation marks, ellipsis, "#", "@", "%", "&", "*", etc.).
* [:graph:]: graphical (human readable) characters, equivalent to [[:alnum:][:punct:]].
* [:print:]: printable characters, equivalent to [[:alnum:][:punct:]\\s].
* [:cntrl:]: control characters (non-printing characters), like `\n` or `\r`.
```

```
str_view_all("ISOM3390", "[:digit:]")
```

```
ISOM3390
```

```
str_view_all("ISOM3390", "\\d")   # need to escape the `\`
```

```
ISOM3390
```

```
str_view_all("ISOM_3390", "\\w")
```

```
ISOM_3390
```

```
str_view_all("ISOM 3390", "[:blank:]")
```

```
ISOM 3390
```

```
str_view_all("ISOM 3390", "[:space:]")
```

```
ISOM 3390
```

```
str_view_all("~!@#$%^&*([{_-+=<>?:;',./", "[:punct:]")
```

```
~!@#$%^&*([{_-+=<>?:;',./
```

Repetition

Quantifiers are used to specify how many times to *repeat the pattern*.

`+` means "1 or more times":

```
str_view(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh+")
```

```
O my gosh!
Oh wow!
Ohhhhh no!
```

`*` means "0 or more times":

```
str_view(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh*")
```

```
O my gosh!
Oh wow!
Ohhhhh no!
```

`?` means "0 or 1 times" (optional once):

```
str_view(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh?")
```

`O` my gosh!

`Oh` wow!

`Oh`hhhh no!

`{n}` means "exactly n times":

```
str_view(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh{3}")
```

O my gosh!

Oh wow!

`Ohhh`hh no!

`{n,}` means "n or more times"; `{n,m}` means "between n and m times" (inclusive):

```
str_view(c("10 dollars", "100 dollars", "1000 dollars"), "10{2,}")
```

10 dollars

`100` dollars

`1000` dollars

```
str_view(c("10 dollars", "100 dollars", "1000 dollars"), "10{1,2}")
```

`10` dollars

`100` dollars

`100`0 dollars

By default, a quantifier applies to the last (meta)character. Use `()` to apply it to a whole group.

```
str_view(c("haaa", "haha"), "ha{2,}")
```

`haaa`

haha

```
str_view(c("haaa", "haha"), "(ha){2,}")
```

haaa

`haha`

Escaping

Metacharacters (`.`, `$`, `^`, `{`, `[`, `(`, `|`, `)`, `]`, `}`, `*`, `+`, `?` and `\`) are special characters that have a special meaning and are not interpreted literally.

If we want to match them literally instead of using their special meaning, we need to use an **escape**.

Like strings, regexps use the backslash `\`, to escape special meaning.

For example, to match a `.`, you need the regexp `\.`. Unfortunately, this creates a problem. Because we use strings to represent regular expressions, and `\` is also used as an escape symbol in strings, so to create the regular expression `\.`, we need the string `"\\."`.

```
writeLines("\\.")
```

```
## \.
```

```
str_view(c("abc", "a.c", "bef"), "a.c")
```

`abc`

`a.c`

bef

```
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

abc

```
a.c
```

```
bef
```

Match a literal `+` with `"\\+"`

```
str_view(c("Business + Programming", "Business - Programming"), "Business \\+|Business -")
```

```
Business + Programming
```

```
Business - Programming
```

We always have to use an **escape sequence** to turn metacharacters into literals.

Question: If `\` is used as an escape character in regular expressions, how do you match a literal `\`?

You need to escape it, creating the regular expression `\\`. To create that regular expression, you need to use a string, which needs to escape both `\`. That means to match a literal `\` you need to write `"\\\\"`. You need four backslashes to match one!

```
writeLines("\\\\")
```

```
## \\
```

```
x <- "a\\b"
writeLines(x)
```

```
## a\b
```

```
str_view(x, "\\\\")
```

```
a\b
```

Anchoring

By default, regexps will match any part of a string, but it is sometimes useful to *anchor a regexp* using an **anchor**.

- `^` (not within square braces) means looking for a match at the **start** of a line.
- `$` means looking for a match at the **end** of a line.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
apple
```

```
banana
```

```
pear
```

```
str_view(x, "a$")
```

```
apple
```

```
banana
```

```
pear
```

Grouping and Backreferences

We can use parentheses `()` to override the default precedence rules.

```
str_view(c("grey", "gray"), "gre|ay")
```

```
grey
```

```
gray
```

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

```
grey
```

```
gray
```

We can also use parentheses `()` to define **groups** that we want to capture and refer to. **Backreferences** (e.g., `\1`, `\2`) refer to the groups by index.

```
str_view_all(c("aaabbcdd", "abbacddd"), "(.)\\1")
```

aaabbcdd

abbacddd

```
str_view_all(c("aaabbcdd", "abbacddd"), "(.)(.)\\2\\1")
```

aaabbcdd

abbacddd

9.4 String Operations Using Regular Expressions

We can use regular expressions in string operations for pattern matching.

For example, we can pass a regular expression (instead of an exact string) to the parameter `pattern` of `str_split()`.

```
str(str_split)
```

```
## function (string, pattern, n = Inf, simplify = FALSE)
```

Splitting with Patterns

```
trump.lines %>% str_split("[:punct:]*[\\s]") %>% .[1:4]
```

```
## [[1]]
##  [1] "Friends"    "delegates"  "and"        "fellow"     "Americans"
##  [6] "I"          "humbly"     "and"        "gratefully" "accept"
## [11] "your"       "nomination" "for"        "the"        "presidency"
## [16] "of"         "the"        "United"     "States."
##
## [[2]]
## [1] "Story"     "Continued" "Below"
##
## [[3]]
## [1] ""
##
## [[4]]
##  [1] "Together"   "we"         "will"       "lead"       "our"
##  [6] "party"      "back"       "to"         "the"        "White"
## [11] "House"      "and"        "we"         "will"       "lead"
## [16] "our"        "country"    "back"       "to"         "safety"
## [21] "prosperity" "and"        "peace"      "We"         "will"
## [26] "be"         "a"          "country"    "of"         "generosity"
## [31] "and"        "warmth"     "But"        "we"         "will"
## [36] "also"       "be"         "a"          "country"    "of"
## [41] "law"        "and"        "order."
```

Instead of splitting up strings by patterns, we can also split up by `character`, `word`, `sentence` and `line`, by setting `boundary()`:

```
str_split("This is a sentence. This is another sentence.", boundary("sentence"))[[1]]
```

```
## [1] "This is a sentence. "      "This is another sentence."
```

```
str_split("This is a sentence. This is another sentence.", boundary("word"))[[1]]
```

```
## [1] "This"     "is"       "a"        "sentence" "This"     "is"       "another"
## [8] "sentence"
```

```
str_split("This is a sentence. This is another sentence.", boundary("character"))[[1]]
```

```
##  [1] "T" "h" "i" "s" " " "i" "s" " " "a" " " "s" "e" "n" "t" "e" "n" "c" "e" "."
## [20] " " "T" "h" "i" "s" " " "i" "s" " " "a" "n" "o" "t" "h" "e" "r" " " "s" "e"
## [39] "n" "t" "e" "n" "c" "e" "."
```

Detecting Matches

To determine if a character vector matches a pattern, use `str_detect()`. It returns a logical vector the same length as the input.

```
str_detect(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh+")
```

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

```
str_detect(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh+") %>% sum()
```

```
## [1] 2
```

Counting Matches

`str_count()` tells you how many matches there are in a string.

```
str_count(c("ISOM3390", "MARK3220"), "[AEIOU]")
```

```
## [1] 2 1
```

Subsetting Matches

Select the elements that ends with a punctuation from `trump.words`:

```
trump.words[str_detect(trump.words, "[:punct:]$")][1:20]
```

```
##  [1] "Friends,"     "Americans:"   "States."      "Together,"    "House,"
##  [6] "safety,"      "prosperity,"  "peace."       "warmth."      "order."
## [11] "nation."      "police,"      "cities,"      "life."        "country."
## [16] "communities." "personally,"  "victims."     "you:"         "end."
```

Alternatively, we can use `str_subset()`, which is a convenient wrapper:

```
trump.words %>% str_subset("[:punct:]$") %>% .[1:20]
```

```
##  [1] "Friends,"     "Americans:"   "States."      "Together,"    "House,"
##  [6] "safety,"      "prosperity,"  "peace."       "warmth."      "order."
## [11] "nation."      "police,"      "cities,"      "life."        "country."
## [16] "communities." "personally,"  "victims."     "you:"         "end."
```

Locating Matches

`str_locate()` and `str_locate_all()` give us the starting and ending positions of the match(es) in a string.

```
x <- c("O my gosh!", "Oh wow!", "Ohhhhh no Oh!")
str_locate(x, "Oh+")       # return an integer matrix
```

```
##      start end
## [1,]    NA  NA
## [2,]     1   2
## [3,]     1   6
```

```
str_locate_all(x, "Oh+")   # return a list of integer matrices
```

```
## [[1]]
##      start end
##
## [[2]]
##      start end
## [1,]     1   2
##
## [[3]]
##      start end
## [1,]     1   6
## [2,]    11  12
```

Extracting Matches

We can first locate the matches and then use `str_sub()` to extract them:

```
str_sub(x, str_locate(x, "Oh+"))
```

```
## [1] NA       "Oh"     "Ohhhhh"
```

Alternatively, we can use `str_extract()`:

```
str_extract(x, "Oh+")
```

```
## [1] NA       "Oh"     "Ohhhhh"
```

`str_extract()` extracts the first match, whereas `str_extract_all()` extracts all matches.

```
str_extract(c("I hate broccoli", "I hate HATE HATE broccoli"), "(hate|HATE)")   # return a vector
```

```
## [1] "hate" "hate"
```

```
str_extract_all(c("I hate broccoli", "I hate HATE HATE broccoli"), "(hate|HATE)")  # return a list of vectors
```

```
## [[1]]
## [1] "hate"
##
## [[2]]
## [1] "hate" "HATE" "HATE"
```

Replacing Matches

`str_replace()` and `str_replace_all()` allow us to replace matches with new strings.

The simplest use is to replace a pattern with a *fixed string*:

```
str_replace(c("apple", "pear", "banana"), "[aeiou]", "-")
```

```
## [1] "-pple"  "p-ar"   "b-nana"
```

```
str_replace_all(c("apple", "pear", "banana"), "[aeiou]", "-")
```

```
## [1] "-ppl-"  "p--r"   "b-n-n-"
```

`str_replace_all()` can perform multiple replacements by supplying a *named vector*:

```
str_replace_all(c("1 apple", "1 apple 2 pears"), c("[aeiou]" = "-", "1" = "one", "2" = "two"))
```

```
## [1] "one -ppl-"           "one -ppl- two p--rs"
```

Instead of replacing with a fixed string, we can use *backreferences* to insert components of the match:

```
trump.lines[1:2]
```

```
## [1] "Friends, delegates and fellow Americans: I humbly and gratefully accept your nomination for the presidenc
y of the United States."
## [2] "Story Continued Below"
```

```
trump.lines[1:2] %>% str_replace("([^ ]+) ([^ ]+) ([^ ]+)", "\\1 \\3 \\2")
```

```
## [1] "Friends, and delegates fellow Americans: I humbly and gratefully accept your nomination for the presidenc
y of the United States."
## [2] "Story Below Continued"
```

[Task 2: Shakespeare's Works]

Project Gutenberg (http://www.gutenberg.org) offers over 50,000 free online books, especially old books (classic literature), for which copyright has expired. We're going to look at the complete works of William Shakespeare, taken from the Project Gutenberg website.

Read in the **shakespeare.txt** file with `read_lines()` and call the result `shakespeare.lines`.

```
shakespeare.lines <- read_lines("shakespeare.txt")
shakespeare.lines[1:5]
```

```
## [1] ""
## [2] "Project Gutenberg's The Complete Works of William Shakespeare, by"
## [3] "William Shakespeare"
## [4] ""
## [5] "This eBook is for the use of anyone anywhere in the United States and"
```

This is a vector of strings, each element representing a "line" of text.

a. Queries about `shakespeare.lines`:

1. How many lines are there?
2. How many characters are there in the longest line?
3. What is the average number of characters per line?
4. How many lines are there with zero characters (empty string)?

Tips: each of these queries should only require one line of code; for the last one, use a logical vector and `sum()`.

b. Remove all empty lines from `shakespeare.lines`. Collapse the lines in `shakespeare.lines` into one big string, separating each line by a space in doing so, using `str_c()`. Call the resulting string `shakespeare.all`. How many characters does this string have?

c. Split up `shakespeare.all` into words, using `str_split()` with `pattern = " +"`. Call the resulting string vector (note: here we are asking you for a vector, not a list) `shakespeare.words`. Use the `unique()` function to compute and store the unique words as `shakespeare.words.unique`. How many unique words are there?

d. Print the top 5 longest words in `shakespeare.words.unique`. Do you recognize any of these as actual words?

Tips: the `order()` function returns the indices that put the vector into increasing order. It can take `decreasing = TRUE` as an argument, to sort/find indices according to decreasing order.

e. Using `table()`, compute counts for the words in `shakespeare.words`, and save the result as `shakespeare.wordtab`. Use name indexing to answer: how many times does the word "rumour" appear? The word "gloomy"? The word "assassination"?

f. Sort `shakespeare.wordtab` so that its entries (counts) are in decreasing order, and save the result as `shakespeare.wordtab.sorted`. Print the 25 most commonly used words, along with their counts.

[End of Task 2]

[Task 3: Shakespeare's Works, Continued]

a. There are a couple of issues with the way we've built our words in `shakespeare.words`.

The first one is that capitalization matters; for example, "and" and "And" are counted as different words.

The second issue is that many words contain punctuation marks (and so, aren't really words in the first place); for example, "and," is counted as a word.

```
# shakespeare.wordtab[c("and", "And", "and,")]
```

Fix these two issues. Start from `shakespeare.all`, split it into words, call the resulting string vector `shakespeare.words.new`. Then, delete all empty strings from this vector, and compute word table from it, called `shakespeare.wordtab.new`.

Tips: `pattern = [[:space:]]|[[:punct:]]` for `str_split()`.

b. Compute the unique words in `shakespeare.words.new`, call the result `shakespeare.words.new.unique`. Then print the top 5 longest words in `shakespeare.words.new.unique`.

c. Sort `shakespeare.wordtab.new` so that its entries (counts) are in decreasing order, and save the result as `shakespeare.wordtab.sorted.new`. Print out the 25 most common words and their counts.

[End of Task 3]