Section 7

Character Functions and Manipulation in R

Character Functions

- R is not known for its prowess in dealing with textual analysis and natural language processing, but it does have some useful features and functions that give it some of the textual functionality of Python and Perl
- Although there are a set of base R functions available to us for textual analysis, we will focus on the `stringr` package, a part of the `tidyverse`
- All functions from the `stringr` package begin with `str_`

base::length()

- Be sure to thoroughly understand the `length()` function, whose results can be unexpected or confusing (or both), without throwing any warning
- For vectors, matrices, arrays, factors, data frames and lists, `length()` returns the **number of elements**

```
> myStr_01 <- "abcdef"
> length(myStr_01)
[1] 1

> myStr_02 <- "abcdefghijklmnopqrstuvwxyz"
> length(myStr_02)
[1] 1

> myVec_01 <- 1:5
> length(myVec_01)
[1] 5
```

stringr::str_length()

- The vectorized `str_length()` function will return the number of characters in a character value

```
> myStr_01 <- "abcdef"
> str_length(myStr_01)
[1] 6

> myStr_02 <- "abcdefghijklmnopqrstuvwxyz"
> str_length(myStr_02)
[1] 26

> myVec_02 <- 12:8
> str_length(myVec_02)
[1] 2 2 2 1 1
```

- Note that `str_length()` coerces numeric values to characters

stringr::str_c

- The `str_c()` function accepts a vector and joins elements together
- To control how the elements are separated, use the `sep=` argument

```
> x <- 99

> str_c('My age is: ', 1 + x, "...boy am I old")
[1] "My age is: 100...boy am I old"

> str_c('My age is: ', 1 + x, "...boy am I old", sep = "<><><>")
[1] "My age is: <><><>100<><><>...boy am I old"
```

stringr::str_c [CONT'D]

- If you pass a **vector** to `str_c`, the `collapse=` argument can be used to collapse the entire vector into a single string, as well as specify a character string to place between each element of the vector

```
> str_c(c("abc", "abcdef", "ZzZzZZzZzZ"))
[1] "abc"        "abcdef"     "ZzZzZZzZzZ"

> str_c("abc", "abcdef", "ZzZzZZzZzZ")
[1] "abcabcdefZzZzZZzZzZ"

> str_c(c("abc", "abcdef", "ZzZzZZzZzZ"), collapse = "")
[1] "abcabcdefZzZzZZzZzZ"

> str_c(c("abc", "abcdef", "ZzZzZZzZzZ"), collapse = " ")
[1] "abc abcdef ZzZzZZzZzZ"
```

stringr::str_c [CONT'D]

- When multiple arguments are passed to `str_c`, it will vectorize the operations, recycling shorter elements when necessary

```
> str_c("x", 1:5, sep = "_")
[1] "x_1" "x_2" "x_3" "x_4" "x_5"
```

stringr::str_sub()

- The `str_sub()` function can be used to extract parts of character strings
- `str_sub()` accepts the arguments `start=` and `end=`, identifying the location of the first and last character (with an integer), respectively, in the the string (inclusive)
- `start=` and `end=` may be omitted
- Negative numbers count backwards from end

```
> (myStr_03 <- str_c(LETTERS[1:8], letters[1:8], collapse = ""))
[1] "AaBbCcDdEeFfGgHh"

> str_sub(myStr_03, start = 6, end = 12)
[1] "cDdEeFf"

> myNum <- 123456789
> str_sub(myNum, start = 3)
[1] "3456789"

> str_sub(myNum, start = -3, end = -1)
[1] "789"
```

stringr::str_sub() [CONT'D]

- str_sub() is vectorized
  - for start= and end= arguments

```
> (myStr_04 <- c("paul", "john", "sally"))
[1] "paul"  "john"  "sally"

> str_sub(myStr_04, 3, 4)
[1] "ul" "hn" "ll"
```

  - and for start= and end= vectors passed to str_sub()

```
> myStr_05 <- c('abcdefg', '12345678', 'XYZ')

> str_sub(myStr_05, start = c(1, 4, 2), end = c(2, 5, 3))
[1] "ab" "45" "YZ"
```

stringr::str_sub() [CONT'D]

- str_sub() may also be used to change the values of part of character strings

```
> myStr_06 <- "my big dog"

> str_sub(myStr_06, 8, 10) <- "cat"

> myStr_06
[1] "my big cat"

> str_sub(myStr_06, 4, 6) <- "gigantic"

> myStr_06
[1] "my gigantic cat"
```

stringr::str_sort()

- A sorting function intentionally designed to sort text
- Accepts a `locale=` argument to follow sorting rules for specific languages and/or locations
- If no `locale` is provided, R uses current `locale` as provided by your OS

```
> (str_sort(letters[1:26], locale = "en"))
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
[12] "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
[23] "w" "x" "y" "z"

> (str_sort(letters[1:26], locale = "haw"))
 [1] "a" "e" "i" "o" "u" "b" "c" "d" "f" "g" "h"
[12] "j" "k" "l" "m" "n" "p" "q" "r" "s" "t" "v"
[23] "w" "x" "y" "z"
```

Other Useful `stringr` Functions

- `str_to_lower()`
- `str_to_upper()`
- `str_trim()`
- The `stringr` package contains 42 functions

Regular Expressions in R

- Regular expressions are a method of expressing patterns in character values which can then be used to extract parts of strings or to modify those strings in some way
- Regular expressions can range from simple to the highly complex
- `stringr` contains a set of functions that facilitate the use of regular expressions to match patterns in text

A Brief Introduction to Regular Expressions

- Regular expression are composed of three components
  1. literal characters, which are matched by a single character
  2. character classes, which can be matched by any number of characters
  3. modifiers, which operate on literal characters or character classes
- As many punctuation marks are regular expression modifiers, the following characters must always be preceded by a backslash to retain their literal meaning (we will come back to this shortly)

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

Literal Character Matches

- Although `str_view()` is a great learning/testing tool, in implementation, use
  1. `stringr::str_detect()`, which will return `logical` values
  2. `stringr::str_subset()`, which returns the actual values
- To find all strings in a vector that contain a literal character match, pass the regular expression to `str_detect()` or `str_subset()`

```
> myVec <- c("my", "name", "is", "paul")

# which strings contain at least one occurrence of the letter "a"
> str_detect(myVec, "a")
[1] FALSE  TRUE FALSE  TRUE

> str_subset(myVec, "a")
[1] "name" "paul"
```

Leftmost Matching

- A regular expression engine always returns the leftmost match, even if a more 'desirable' match can be found further to the right in a string
- A regex engine examine the first character in a string and, only when all possibilities are exhausted, does it move on to the next character in the string
- This behavior results in the regex engine always returning the leftmost match, never proceeding further once a match is made

```
> myVec <- "the dogmatic dog is dogged about her dogma"

> str_view(myVec, "dog")
```

the dogmatic dog is dogged about her dogma

A Brief Digression: `str_view_all` and `str_count`

- `str_view_all` identifies all matches with a string, not only the first/leftmost match

```
> myVec <- c("my lazy summer by the pool")

> str_view_all(myVec, "y")
```

my lazy summer by the pool

- `str_count` will count the number of time a match is made in a string

```
> myVec <- "the dogmatic dog is dogged about her dogma"

> str_count(myVec, "dog")
[1] 4
```

Modifiers for Regular Expressions

Modifier	*Meaning*
^	anchors expression to the beginning of target
$	anchors expression to end of target
.	matches any single character except newline
|	separates alternative patterns
()	groups patterns together
*	matches 0 or more occurrences of preceding entity
?	matches 0 or 1 occurrence of preceding entity
+	matches 1 or more occurrences of preceding entity
$\{n\}$	matches exactly n occurrences of preceding entity
$\{n,\}$	matches n or more occurrences of preceding entity
$\{n,m\}$	matches between n and m occurrences of preceding entity

Modifiers [EXAMPLES]

```
> myVec <- c("Mississippi", "Massachusetts", "Connecticut")

> str_subset(myVec, "^M")
[1] "Mississippi"   "Massachusetts"

> str_subset(myVec, "t$")
[1] "Connecticut"

> str_subset(myVec, "M...i")
[1] "Mississippi"

> str_subset(myVec, "n?")
[1] "Mississippi"   "Massachusetts" "Connecticut"

> str_subset(myVec, "s{1}")
[1] "Mississippi"   "Massachusetts"

> str_subset(myVec, "ss{1}")
[1] "Mississippi"   "Massachusetts"

> str_subset(myVec, "iss{2}")
character(0)

> str_subset(myVec, "(iss){2}")
[1] "Mississippi"
```

Forming Character Classes

- A character class instructs the regex engine to to match only one of the characters in a class
- To form a character class, use square brackets, [ ], to surround the characters to be matched

E.g. to create a character class that will be matched either by `x` OR `y` OR `z` OR the number `1`, use `[xyz1]`

```
> myVec <- c("my", "name", "is", "paul")

# which strings contain at least one occurrence of the letter "a",
#    followed by EITHER the letter "m" OR the letter "u"?

> str_subset(myVec, "a[mu]")
[1] "name" "paul"
```

- Observe that square brackets are not interpreted literally in this context; if you are searching for a literal square bracket character, it must be preceded by two backslashes

Forming Character Classes [CONT'D]

- Dashes are used inside of character classes to represent a range of values, e.g., `[a-z]`, `[3-7]` or `[a-zA-Z0-9]`
- If a dash is to be literally included in a character class, it should be either the first character in the class `OR` it should be preceded by a backslash
- Other special characters, save square brackets, do not need to be preceded by a backslash when used in a character class
- When searching for a digit, instead of passing the character class `[0-9]`, an equivalent shortcut is to pass `\\d`, i.e., `[0-9]` $\equiv$ `\\d`

Forming Character Classes [EXAMPLES]

```
> myVec <- c("my", "PHONE", "number", "is", "+1", "123-456-7890")

# which strings contain at least one occurrence of "A" through
#    "Z" or the numbers "0" through "9"?
> str_subset(myVec, "[A-Z0-9]")
[1] "PHONE"        "+1"           "123-456-7890"

# which strings contain at least one occurrence of a dash or "A" through "Z"?
> str_subset(myVec, "[-A-Z]")
[1] "PHONE"        "123-456-7890"

# why does this fail?
> str_subset(myVec, "[A-Z\-]")
Error: '\-' is an unrecognized escape in character string starting ""[A-Z\-"

# why doesn't this fail?
> str_subset(myVec, "[A-Z\\-]")
[1] "PHONE"        "123-456-7890"

# in a character class, special characters (except dash) do not need to be escaped,
#     and can be located anywhere within the class
> str_subset(myVec, "[+A-Z]")
[1] "PHONE" "+1"

> str_subset(myVec, "[A-Z+]")
[1] "PHONE" "+1"
```

Negated Character Classes

- Typing a caret, `^`, after the opening square bracket of a character class negates the class
- **n.b.** A negated character class still must match a character
- `r[^t]`
  - **MEANS** an `r` proceeded by a character that is not a `t`
  - **DOES NOT MEAN** an `r` not proceeded by a `t`

```
> myVec <- c("zipper", "your zipper is down")

> str_view(myVec, "r[^t]")
```

```
zipper
your zipper is down
```

Modifiers **Inside** Character Classes

- There are four modifiers which need to be carefully accounted for when being used inside a character class in regular expressions: `]`, `^`, `\`, and `-`
- Other modifiers do not need to be accounted for in any special way, e.g., when searching for a literal `+` or `?`, the character class `[+?]` can be used

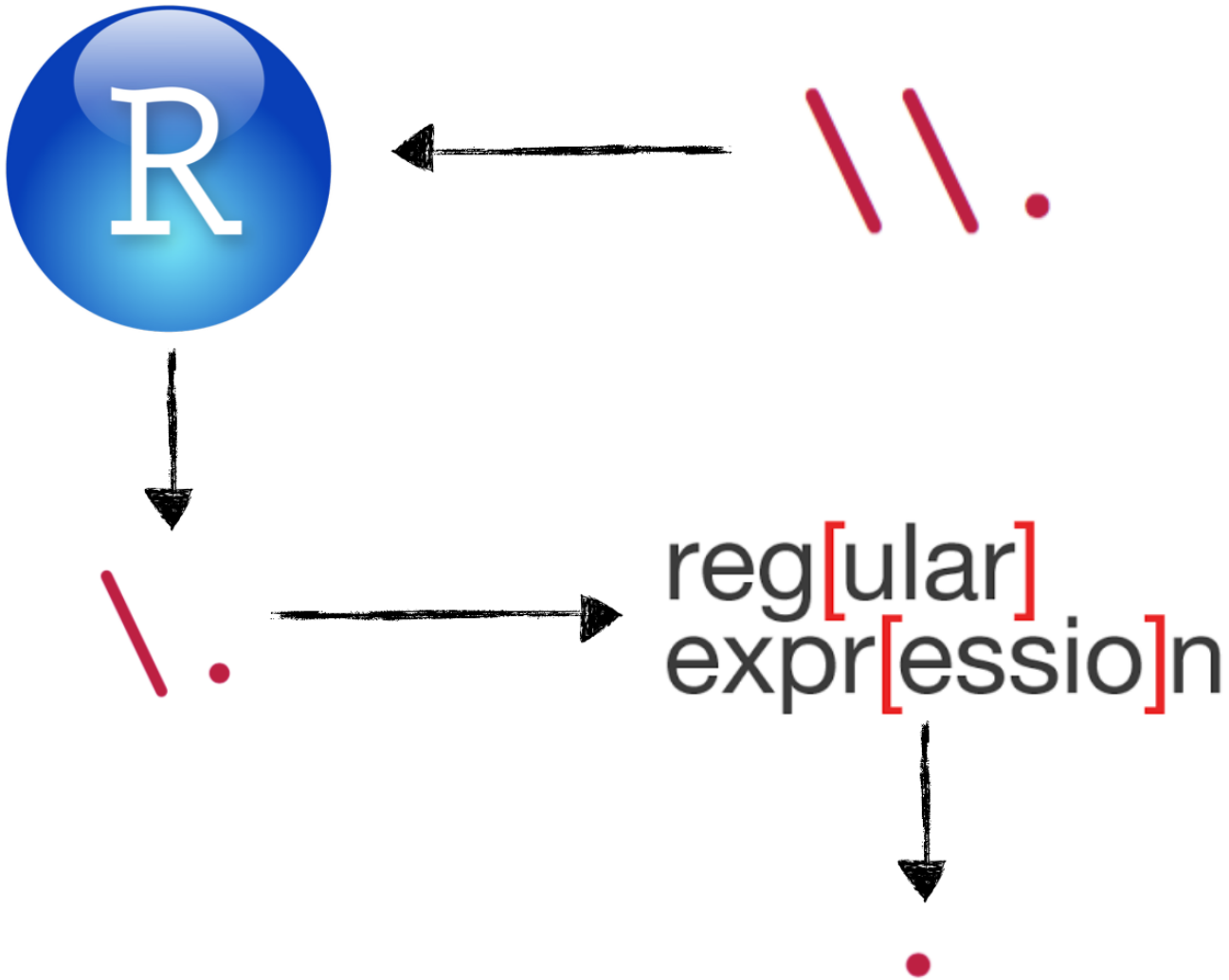
The Backslash and Regular Expressions

- Regular expressions have specific modifiers that can be used to to change the behavior of the regular expression
- We have already seen the square bracket

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

- This leaves us with the problem of being able to literally identify one of these modifiers in a string
- To achieve this, there are two required steps
  1. To inform the regular expression engine, i.e., the engine that parses the regular expression, that a literal modifier needs to be parsed, we need to precede that literal with a \
  2. But \ is also an escape character for R, therefore we need to add an additional \
- In sum, to search for a literal ., pass the expression \\.

The Backslash and Regular Expressions [IN REVERSE]



\: Modifiers **Inside** Character Classes

- The backslash, \, may be the most complex literal to search for in R
- Searching for a literal \ requires passing four backslashes, \\\\, to the regular expression

```
# to write a string with a backslash, you are required to include double
#   backslashes in the R string as the backslash is itself an escape character in R
> myVec <- c("c:\\myHDD\\paul", "noSlashes")

> str_view(myVec, "\\\\")
```

```
c:\myHDD\paul
noSlashes
```

^: Modifiers **Inside** Character Classes

- The caret, `^`, **if** placed directly proceeding the opening square bracket of a character class, will negate the proceeding character or class of characters
- There caret can be placed anywhere else within the character class and will be interpreted literally, i.e., without having to be escaped with a backslash
- If the caret is the leading literal character in a character class passed to a regular expression, it must be escaped with a `\\`

```
> myVec <- c("x^2", "aaa", "hats^^^")

> str_subset(myVec, "[^a]")
[1] "x^2"     "hats^^^"

> str_subset(myVec, "[\\^]")
[1] "x^2"     "hats^^^"

> str_subset(myVec, "s[abcx^]")
[1] "hats^^^"
```

]: Modifiers **Inside** Character Classes

- The closing square bracket,], must be escaped with \\ to be detected as a literal character

```
> myVec <- c("[-1-]", "[[[", "aBCd")

> str_subset(myVec, "[\\[]")
[1] "[-1-]" "[[["
```

-: Modifiers **Inside** Character Classes

- The hyphen, `-`, placed directly proceeding an opening square bracket for a class or directly preceding a closing square bracket for a class, will be interpreted as a literal character
- It is used between two characters to create a class
- In an any other location, it must be escaped with a `\\` to be interpreted as a literal character

```
> myVec <- c("[-1-]", "[[[", "aBCd")

> str_subset(myVec, "[-a-z]")
[1] "[-1-]" "aBCd"

> str_subset(myVec, "[a-z-]")
[1] "[-1-]" "aBCd"

> str_subset(myVec, "[-]")
[1] "[-1-]"

> str_subset(myVec, "[b\\-z]")
[1] "[-1-]"
```

Literal Searches for Other Modifiers

```
> myVec <- c("what is your name?", "f_name.l_name")

> str_subset(myVec, "\\.")
[1] "f_name.l_name"

> str_subset(myVec, "\\?")
[1] "what is your name?"
```

Character Class Subtraction

- Matches any character present in a class that is not in the subtracted class

E.g. To match a single letter that is not a vowel

```
> myVec <- c("bcd", "aei", "ux")

> str_subset(myVec, "[[a-z]-[aeiou]]")
[1] "bcd" "ux"

> str_subset(myVec, "[b-df-hj-np-tv-z]")
[1] "bcd" "ux"
```

Character class subtraction is more language-sensitive than most regular expressions, so take care when adapting these expressions in a different language

Character Class Shortcuts

Shortcut	*Matches*
`\\d`	`[0-9]`
`\\D`	`[^\\d]`
`\\w`	`[a-zA-Z0-9_]`
`\\W`	`[^\\w]`
`\\s`	`[ \t\r\n\f]`
`\\S`	`[^\\s]`

- `[ \t\r\n\f]` ≡ space, tab, line break or form feed
- `\\s` is particularly sensitive to choice of language so beware

WHY? `[^\d\s]` ≠ `[\D\S]`

Modifiers: .

- The dot, ., matches a single character, except for the a line break character
- The dot is a powerful modifier; it allows you to be lazy, but use it cautiously... (pun intended)

Modifiers: ^ and $

- These modifiers do not match characters, rather, they identify a **position** in the regular expression
- ^ will anchor a regular expression to the **beginning** of a string, i.e., it matches the position before the the first character in a string

n.b. do not confuse this with using a leading ^ in a character class, which negates the class

- $ will anchor a regular expression to the **end** of a string, i.e., it matches the position after the last character in a string

```
> myVec <- c("abc123xyz", "999")

> str_subset(myVec, "\\d\\d\\d")
[1] "abc123xyz" "999"

> str_subset(myVec, "^\\d\\d\\d$")
[1] "999"
```

Modifiers: |

- The | modifier, representing alternation, permits the matching of one out of many regular expressions

E.g., Searching a string for either the words `boy` or `girl` or `child`, use the regular expression `boy|girl|child`; in English, this reads as '*find either boy OR girl OR child*' (with some caveats)

- The alternation modifier will select **everything** to the left of the | or **everything** to the right
- To search **exclusively** for the words `boy` or `girl`, one should use groupings, `()` and word boundaries (coming soon)

```
> myVec <- c("mygirl", "boy", "girlboy", "gir")

> str_view(myVec, "girl|boy")
[1] "mygirl"   "boy"    "girlboy"
```

Modifiers: ?

- The ? modifier allows for **optional** items to be evaluated, i.e., items that can be present but are not necessarily present
- The ? modifier allows for an **optional** string to occur **zero or once**
- Proceed the optional letter(s) with the ? modifier
- Classical examples include the difference between American and British, where British English often includes an additional 'u' in many words, or the shortening of dates

```
> myVec <- c("Dec", "neighbour", "neighbor")

> str_subset(myVec, "neighbou?r")
[1] "neighbour" "neighbor"

> str_subset(myVec, "Dec(ember)?")
[1] "Dec"
```

Modifiers: *, + and {}

- The * modifier allows for **optional** items to be evaluated, i.e., items that can be present but are not necessarily present
- The * modifier allows for an **optional** string to occur **zero or more** times
- The + modifier allows for repeated items to occur **one or more** times
- The {} modifier allows for repeated items to evaluated $\{min, max\}$ times

```
> myVec <- "<HTML tag>"

> str_subset(myVec, "<[A-Za-z][A-Za-z]*")
[1] "<HTML tag>"
```

Modifiers: ()

- By placing part of a regular expression inside (), that part of the regular expression is now considered a group
- When grouped, modifiers can be applied to the entire group

n.b. [] defines a character class, () defines a group

```
> myVec <- "<HTML tag>"

> str_subset(myVec, "<[A-Za-z][A-Za-z]*")
[1] "<HTML tag>"
```

Word Boundaries: \\b

- Word boundaries facilitate the isolation and identification of individual strings in a string
- To identify the whole word 'is' in a string the word 'is' should be preceded and proceeded by a \\b

```
> myVec <- "This string is a sample"

> str_view(myVec, "\\bis\\b")
```

This string is a sample

Modifiers for Regular Expressions [EXAMPLE]

- Modifiers operate on whatever entity then follow, using parentheses for grouping if necessary

E.g. To identify a string with two digits, followed by one or more uppercase or lowercase letters, the matching regular expression would be
'[0-9][0-9][a-zA-Z]+'

```
> myVec <- c("Mississippi", "Massachusetts", "Connecticut")

> str_subset(myVec, "\\.")
[1] "f_name.l_name"

> str_subset(myVec, "\\?")
[1] "what is your name?"
```

Modifiers for Regular Expressions [EXAMPLE] [CONT'D]

E.g. For `jpg` filename consisting exclusively of letters, the matching regular expression could be
'`^[a-zA-Z]+\\.jpg$`'

- Observe how this regular expression is constructed
  - `^` explicitly states that the file must **begin** with whatever proceeds it, in this case, `[a-zA-Z]`
  - `+` allows for any number of letters, so long as there is at least one
  - The second backslash, i.e., the backslash on the right, is required so that `.` can retain its literal meaning (recall `.` is a regular expression modifier
  - The first backslash, i.e., the backslash on the left, is required for R
  - `$` ensures that the **final** four characters in the file name are `.jpg`

Even Regular Expressions can be Greedy

- * is **greedy**

```
> myVec <- '"abc"'

> str_view(myVec, "\".*\"")
```

"abc"

```
> myVec <- '"abc" is my favorite "quote"'

> str_view(myVec, "\".*\"")
```

"abc" is my favorite "quote"

- The + is similarly greedy

Even Regular Expressions can be Greedy [CONT'D]

- ? is **greedy**
- The ? can be made to be **lazy** by including a second question mark after the first

```
> myVec <- "my favorite color is red"

> str_view(myVec, "fav(orite)?")
```

```
my favorite colour is red
```

```
> myVec <- "my favorite color is red"

> str_view(myVec, "fav(orite)??")
```

```
my favorite colour is red
```

- The * and {} can also be made lazy by adding a ?

str_split()

- The `str_split()` function can use a character string or regular expression to divide up a character string into smaller pieces
- `str_split()` returns its results in a list, regardless of input
- To break up a sentence into its constituent words

```
> myString_07 <- "I enjoy reading books"

> str_split(myString_07, "")
[[1]]
 [1] "I" " " "e" "n" "j" "o" "y" " " "r" "e" "a" "d" "i" "n" "g"
[16] " " "b" "o" "o" "k" "s"

> str_split(myString_07, " ")
[[1]]
[1] "I"       "enjoy"   "reading" "books"
```

str_split() with Regular Expressions

- Given `str_split()` accepts regular expressions to determine where to split a string, the function is very versatile

E.g. It commonly occurs that when customers input free-form text on surveys, they may accidentally include more than once space between words, thereby requiring a regular expression to appropriately handle the variable inputs

```
> myString_08 <- "I     enjoy reading   books"

> str_split(myString_08, " ")
[[1]]
 [1] "I"       ""        ""        ""        ""        "enjoy"
 [7] "reading" ""        ""        "books"

> str_split(myString_08, " +")
[[1]]
[1] "I"       "enjoy"   "reading" "books"
```

str_locate

- str_locate pinpoints and can extract those parts of a string that are matched by a regular expression
- str_locate outputs a matrix of *starting and erneding positions* of the regular expressions found; if none are found, NAs are returned
- str_locate only provides information on the **first** match in a given input string

```
> myVec <- c("94112 94117", "H8P 2S5", " 90210", "47907-1233")

> str_locate(myVec, "\\d{5}")
     start end
[1,]     1   5
[2,]    NA  NA
[3,]     2   6
[4,]     1   5
```

str_locate_all

- str_locate_all operates similarly to str_locate, but returns information on **all** matches found (not just the first)
- str_locate_all always returns its result in the form of a list

```
> myVec <- c("94112 94117", "H8P 2S5", " 90210", "47907-1233")

> str_locate_all(myVec, "\\d{5}")
[[1]]
     start end
[1,]     1   5
[2,]     7  11

[[2]]
     start end

[[3]]
     start end
[1,]     2   6

[[4]]
     start end
[1,]     1   5
```

Substitutions

- To substitute text based on regular expressions, `stringr` provides two functions, `str_replace` and `str_replace_all`
- Both functions accept
  1. a regular expression
  2. a string containing what will be substituted for the regular expression
  3. a string (or strings) to operate on
- `str_replace` changes **only** the first occurrence of the regular expression, whereas `str_replace_all` changes all occurrences

```
> myVec <- c("paul", "steven", "cynthia")

> str_replace(myVec, "[aeiou]", "#")
[1] "p#ul"    "st#ven"  "cynth#a"

> str_replace_all(myVec, "[aeiou]", "#")
[1] "p##l"    "st#v#n"  "cynth##"
```

str_replace_all

- A common application of `str_replace_all` is the scrubbing of financial data

```
> financialData_01 <- c("$11,345.65", "$99,125.22", "$13,321.99")

> str(financialData_01)
 chr [1:3] "$11,345.65" "$99,125.22" "$13,321.99"

> as.numeric(financialData_01)
[1] NA NA NA
Warning message:
NAs introduced by coercion

> str_replace(financialData_01, "[$,]", "")
[1] "11,345.65" "99,125.22" "13,321.99"

> str_replace_all(financialData_01, "[$,]", "")
[1] "11345.65" "99125.22" "13321.99"
```

- The `str_replace_all` function replaces all instances of `$` and `,` with nothing (''), effectively deleting them from the string

Jan Goyvaerts

https://www.regular-expressions.info