Topic 9: String Operations

ISOM3390: Business Programming in R

Trump's Tweets

> I want to do negative ads on John Kasich, but he is so irrelevant to the race that I don’t want to waste my money.
>
> — **Donald J. Trump (@realDonaldTrump)** 9:05 AM - Nov 20, 2015

```
trump.tweets  # Use the DT package to display data via the DataTables JavaScript library.
```

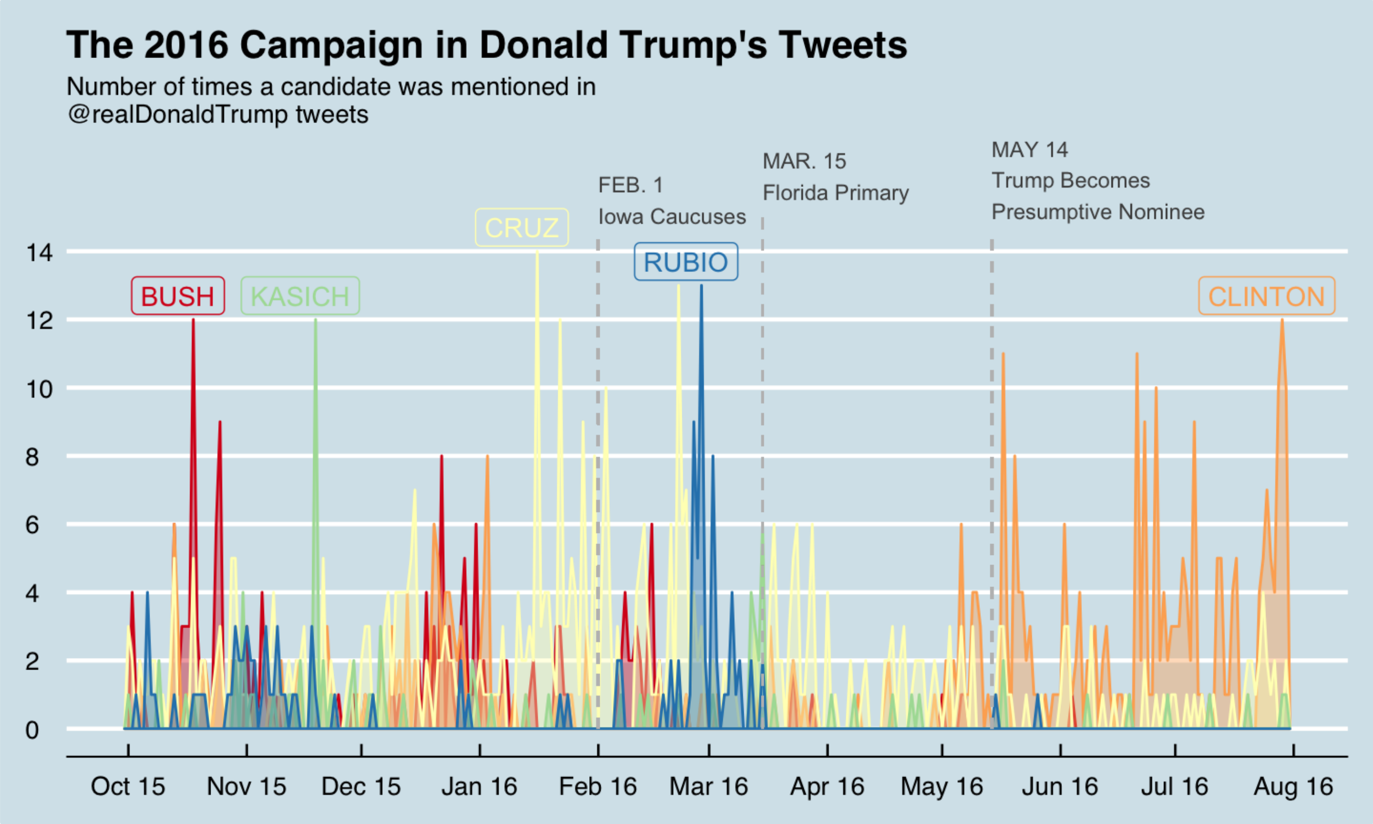
Tweet ID	Text	Date	Favorites	Retweets
759592590106849280	Nielson Media Research final numbers on ACCEPTANCE SPEECH: TRUMP 32.2 MILLION. CLINTON 27.8 MILLION. Thank you!	2016-07-30T23:32:40Z	13850	4130
759524001613918208	Thank you to all of the television viewers that made my speech at the Republican National Convention #1 over Crooked Hillary and DEMS.	2016-07-30T19:00:07Z	27659	6842
759516008272932864	Can you imagine if I had the small crowds that Hillary is drawing today in Pennsylvania. It would be a major media event! @CNN @FoxNews	2016-07-30T18:28:22Z	19968	6488
759515080010719232	NATO commander agrees members should pay up via @dcexaminer:http://www.washingtonexaminer.com/nato-commander-agrees-members-should-pay-up/article/2598183?custom_click=rss …	2016-07-30T18:24:40Z	11624	4668
759513644258525184	Wow, NATO's top commander just announced that he agrees with me that alliance members must PAY THEIR BILLS. This is a general I will like!	2016-07-30T18:18:58Z	23922	7819

Showing 1 to 5 of 4,654 entries

Previous 1 2 3 4 5 … 931 Next

What's in Between?

```
trump.tweets %>% ... %>%
  ggplot(aes(x = Date, y = Count, fill = Candidate, colour = Candidate)) + geom_polygon()
```



Why Do We Care?

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?

The simplest distinction:

- **Character:** a symbol in a written language, like letters, numerals, punctuation, space, etc.
- **String:** a sequence of characters bound together

```r
typeof("r")
```

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

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

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

How to Make Strings?

Just use double quotes or single quotes and type anything in between:

```
(str.1 <- "Business")
```

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

```
(str.2 <- 'Programming')
```

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

We often prefer double quotes to single quotes, because then we can use apostrophes.

```
(str.3 <- "isn't that bad")
```

```
## [1] "isn't that bad"
```

To include a literal single or double quote in a string, we use `\` to **"escape"** it:

```
(double_quote <- "\"")  # or '\"'
```

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

```
(double_quote <- '\'')  # or "\'"
```

```
## [1] "'"
```

Printing Strings

The printed representation of a string is not the same as string itself, because the printed representation shows the escapes.

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

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

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

```
writeLines(x)
```

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

Note: Because `\` is used as the escape, we need to double it up `\\` to include a literal backslash.

Whitespaces

Whitespaces (`?'"'` or `?"'"` to see the complete list) includes:

- `" "` for space
- `"\n"` for newline
- `"\t"` for tab

They count as characters and can be included in strings:

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

```
## [1] "Dear Students,\n\nWelcome to ISOM3390!\n\nSincerely, Justin"
```

```r
writeLines(message)  # or cat(message)
```

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

Converting Other Data Types to Strings

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

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

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

```
as.character(8e+09)
```

```
## [1] "8e+09"
```

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

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

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

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

Converting Strings to Other Data Types

Depends on the given string, of course:

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

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

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

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

```
as.numeric("0.5e-10")
```

```
## [1] 5e-11
```

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

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

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

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

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

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

```
as.numeric(c("0.5", "TRUE"))
```

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

Introduction to `stringr`

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 Ruby or 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 `%>%`.

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

```
##  [1] "str_c"           "str_conv"        "str_count"       "str_detect"      "str_dup"         "str_extract"
##  [7] "str_extract_all" "str_flatten"     "str_glue"        "str_glue_data"   "str_interp"      "str_length"
## [13] "str_locate"      "str_locate_all"  "str_match"       "str_match_all"   "str_order"       "str_pad"
## [19] "str_remove"      "str_remove_all"  "str_replace"     "str_replace_all" "str_replace_na"  "str_sort"
## [25] "str_split"       "str_split_fixed" "str_squish"      "str_sub"         "str_sub<-"       "str_subset"
## [31] "str_to_lower"    "str_to_title"    "str_to_upper"    "str_trim"        "str_trunc"       "str_view"
## [37] "str_view_all"    "str_which"       "str_wrap"
```

Can be grouped into four main families:

- Character manipulation functions that manipulate individual characters within the strings.
- Whitespace tools to add, remove, and manipulate whitespace.
- Pattern matching functions that recognize four engines of pattern description. The most common is regular expressions, but there are three other tools.
- Locale sensitive operations whose operations will vary from locale to locale.

Number of Characters

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

```
length("code monkey")
```

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

```
str_length("code monkey")
```

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

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

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

Padding, Trimming, and Truncting Strings

`str_pad()` pads a string to a fixed length by adding extra whitespace on the `left`, `right`, or `both` sides:

```
str_pad("beer", width = 11, side = "both", pad = "!")
```

```
## [1] "!!!beer!!!!"
```

`str_trim()` removes leading and trailing whitespace:

```
x <- c("Text ", "  with", " whitespace ", " on", "both ", " sides ")
rbind(str_trim(x, side = "left"), str_trim(x))
```

```
##      [,1]    [,2]   [,3]          [,4] [,5]    [,6]
## [1,] "Text " "with" "whitespace " "on" "both " "sides "
## [2,] "Text"  "with" "whitespace"  "on" "both"  "sides"
```

`str_trunc()` truncate a character string.

```
x <- "This string is moderately long"
rbind(str_trunc(x, 20, "right"), str_trunc(x, 20, "left"), str_trunc(x, 20, "center"))
```

```
##      [,1]
## [1,] "This string is mo..."
## [2,] "...s moderately long"
## [3,] "This stri...ely long"
```

Getting a Substring

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:

```
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 end
```

```
## [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))
```

Combining Strings

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 have no separator
```

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

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

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

```
str_c("prefix-", c("a", "b", "c"), "-suffix")  # it vectorizes
```

```
## [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

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

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

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

An Example: Trump's Words

```
# Load a text file from the Web
trump.lines <- read_lines("https://drive.google.com/uc?export=download&id=1AY90rBHoMfLJm_ZMk8NlNZK3yMiWU_Hl")
class(trump.lines)  # we have a character vector
```

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

```
length(trump.lines)  # many lines (elements)!
```

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

```
trump.lines[1:3]  # First 3 lines
```

```
## [1] "Friends, delegates and fellow Americans: I humbly and gratefully accept your nomination for the presidency of the United States."
## [2] "Story Continued Below"
## [3] ""
```

Make one long string:

```
(trump.text<- trump.lines %>% str_c(collapse = " ")) %>% str_sub(1, 128)
```

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

Splitting up a String into Pieces

Use `str_split()` to split a string up into pieces. Because each component might contain a different number of pieces, this returns a list:

```
trump.words <- str_split(trump.text, " ")
str(trump.words)
```

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

Our most basic tool for summarizing text: **word counts**, retrieved using `table()`:

```
trump.words <- trump.words[[1]]
(trump.wordtab <- table(trump.words))[61:78]
```

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

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? and more …

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?

What are Regular Expressions?

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

Regexps follow a well-defined set of rules, independent of the R language:

- A valid regexp can be a sequence of **literals** (i.e., a string we want to match literally). E.g.:
  - `"fly"` matches "superfly", "why walk when you can fly".
  - It does not match "time flies like an arrow", "fruit flies like bananas".
- `OR` of 2 regexps is a regexp. E.g.,
  - `"fly|flies"` tries to match "fly" or "flies".
- Concatenation of regexps is a regexp. E.g.,
  - `"(time|fruit) (fly|flies)"` tries to match "time" or "fruit", then a space, then "fly" or "flies".
  - Parentheses define groups; more on this later.

Showing Matches to a Regexp

`str_view()` and `str_view_all()` functions take a character vector and a regular expression, and show us how they match with HTML rendering.

```
str.vec <- c("time flies when you're having fun in 3390", "fruit flies when you throw it", "how do you spell fruitfly?")
```

```
str_view(str.vec, "fly")
```

```
• time flies when you're having fun in 3390
• fruit flies when you throw it
• how do you spell fruitfly?
```

```
str_view(str.vec, "fly|flies")
```

```
• time flies when you're having fun in 3390
• fruit flies when you throw it
• how do you spell fruitfly?
```

```
str_view(str.vec, "(time|fruit)(fly|flies)")
```

```
• time flies when you're having fun in 3390
• fruit flies when you throw it
• how do you spell fruitfly?
```

```
str_view(str.vec, "(time|fruit) (fly|flies)")
```

```
• time flies when you're having fun in 3390
• fruit flies when you throw it
• how do you spell fruitfly?
```

Matches never overlap. For example, in `"abababa"`, how many times will the pattern `"aba"` match:

```
str_view_all("abababa", "aba")
```

```
• abababa
```

Many `stringr` functions come in pairs: one function works with a single match, and the other works with all matches. The second function will have the suffix `_all`.

Metacharacters

Metacharacters are special characters that have a special meaning and are not interpreted literally.

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 caret inside square braces negates what follows. E.g., `"[^0-9]"` tries to match anything but a number between 0 and 9

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

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

```
• ISOM3390
• MARK3220
```

- A period `"."` tries to match any character (except a newline; don't even need square braces)

```
str_view(c("ISOM3390", "MARK3220"), ".M.")
```

```
• ISOM3390
• MARK3220
```

Predefined Character Classes

- `[: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.
- `[:graph:]`: graphical (human readable) characters: equivalent to `[[:alnum:][:punct:]]`.
- `[:print:]`: printable characters, equivalent to `[[:alnum:][:punct:]\\s]`.
- `[:cntrl:]`: control characters, like `\n` or `\r`, `[\x00-\x1F\x7F]`.

```
str_view_all("p.u,n;c:t!u?a*t_i&o#n", "[:punct:]")
```

```
• p.u,n;c:t!u?a*t_i&o#n
```

Quantifiers for Repetition

Quantifiers allow us to express: how many times a pattern matches?

- `"+"` 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!
• Ohhhhh no!
```

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

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

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

- `"{n,}"` means "n or more times", and `"{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
• 1000 dollars
```

By default, a quantifier applies to the last (meta)character.

Use `()` to have it apply to a whole group.

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

```
• haaa
• haha
```

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

```
• haaa
• haha
```

Escape Sequences

In regexps, metacharacters (`.`, `$`, `^`, `{`, `[`, `(`, `|`, `)`, `]`, `}`, `*`, `+`, and `?`) have special meaning.

We need to use an **"escape"** to match them literally, instead of using their special behaviours.

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

However, because `\` is also used as an escape symbol in strings, we always have to use an **escape sequence** to turn metacharacters into literals. E.g.:

- `"\\["` tries to match a left square brace

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

```
## \.
```

- `"\\\\"` tries to match a backslash

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

```
## \\
```

```r
str_view(c("Business + Programming = Magic", "Business - Programming = Tedious Coding"), "Business \\+|Business -")
```

```
· Business + Programming = Magic
· Business - Programming = Tedious Coding
```

Anchoring

By default, regexps will match any part of a string.

It's often useful to anchor a regexp using an **anchor**:

- When `"^"` is used outside of square braces, it means looking for a match at the start of a line.
- When `"$"` is used, it means looking for a match at the end of a line.

```
str_view(c("<strong> hi </strong>", "bye </HTML>", "a <b> c </b> d"), "^<.+>|<.+>$")
```

```
• <strong> hi </strong>
• bye </HTML>
• a <b> c </b> d
```

Grouping and Backreferences

Use `()` to define "groups" that we want to capture and refer to for matching.

Backreferences, like `\1`, `\2`, etc., refer to these capturing groups by index.

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

- aaabbcdd
- abbacddd

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

- aaabbcdd
- abbacddd

```
str_view(c("<strong> hi </strong>", "bye </HTML>", "a <b> c </b> d"), "<(.+)>.+</\\1>")  # uses a escape sequence
```

- <strong> hi </strong>
- bye </HTML>
- a <b> c </b> d

Splitting with Patterns

```
trump.lines %>% str_split("[:punct:]*[\\s]") %>% .[1:2]
```

```
## [[1]]
##  [1] "Friends"    "delegates"  "and"        "fellow"     "Americans"  "I"          "humbly"     "and"
##  [9] "gratefully" "accept"     "your"       "nomination" "for"        "the"        "presidency" "of"
## [17] "the"        "United"     "States."
##
## [[2]]
## [1] "Story"     "Continued" "Below"
```

We can use `simplify = TRUE` to return a matrix, and also request a maximum number of pieces using `n`:

```
trump.lines %>% str_split("[:punct:]*[\\s]", n = 10, simplify = TRUE) %>% .[1:2, ]
```

```
##      [,1]      [,2]        [,3]    [,4]     [,5]        [,6] [,7]     [,8]  [,9]
## [1,] "Friends" "delegates" "and"   "fellow" "Americans" "I"  "humbly" "and" "gratefully"
## [2,] "Story"   "Continued" "Below" ""       ""          ""   ""       ""    ""
##      [,10]
## [1,] "accept your nomination for the presidency of the United States."
## [2,] ""
```

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

```
x <- "This is a sentence.  This is another sentence."
str_split(x, " ")[[1]]
```

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

```
str_split(x, boundary("word"))[[1]]
```

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

```
str_split(x, boundary("character"))[[1]]
```

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

```
str_split(x, boundary("sentence"))[[1]]
```

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

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
```

Rather than a simple yes or no, `str_count()` tells you how many matches there are in a string:

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

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

Subsetting Matches

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

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

`str_subset()` is a convenient wrapper:

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

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

Locating Matches

`str_locate()` and `str_locate_all()` give us the starting and ending positions of each match.

```
(sub_position <- str_locate(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), "Oh+"))
```

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

Then we can use `str_sub()` to extract and/or modify them:

```
str_sub(c("O my gosh!", "Oh wow!", "Ohhhhh no!"), sub_position)
```

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

Extracting Matches

To extract the text of the first match, use `str_extract()`.

```
str_extract(c("I hate broccoli", "I hate HATE HATE broccoli, it disgusts me, I hate it"), "(hate|HATE)")
```

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

To get all matches for each string, use `str_extract_all()`, which returns a list:

```
str_extract_all(c("I hate broccoli", "I hate HATE HATE broccoli, it disgusts me, I hate it"), "(hate|HATE)")
```

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

Use `simplify = TRUE`, `str_extract_all()` will return a matrix with short matches expanded to the same length as the longest.

```
##      [,1]   [,2]   [,3]   [,4]
## [1,] "hate" ""     ""     ""
## [2,] "hate" "HATE" "HATE" "hate"
```

`str_match()` gives each individual component. Instead of a character vector, it returns a matrix, with one column for the complete match followed by one column for each group:

```
trump.lines[1:4]
```

```
## [1] "Friends, delegates and fellow Americans: I humbly and gratefully accept your nomination for the presidency 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, prosperity, and peace. We wi
```

```
trump.lines %>% str_match("(a|the) ([^ ]+)") %>% .[1:4, ]
```

```
##      [,1]              [,2]  [,3]
## [1,] "the presidency" "the" "presidency"
## [2,] NA                NA    NA
## [3,] NA                NA    NA
## [4,] "the White"       "the" "White"
```

Use `str_match_all()` if we want all matches for each string:

```
trump.lines %>% str_match_all("(a|the) ([^ ]+)") %>% .[1:4]
```

```
## [[1]]
##      [,1]               [,2]  [,3]
## [1,] "the presidency" "the" "presidency"
## [2,] "the United"     "the" "United"
##
## [[2]]
##      [,1] [,2] [,3]
##
## [[3]]
##      [,1] [,2] [,3]
##
## [[4]]
##      [,1]        [,2]  [,3]
## [1,] "the White" "the" "White"
## [2,] "a country" "a"   "country"
## [3,] "a country" "a"   "country"
```

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 house", "2 cars", "3 people"), c(`1` = "one", `2` = "two", `3` = "three"))

## [1] "one house"    "two cars"     "three people"
```

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

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

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

Working with `dplyr` Verbs

```
## # A tibble: 400 x 2
##    index tweet
##    <int> <chr>
##  1     1 Metaps, Japanese App Monetization firm, raises $36M series C to push Into #BigData, #MachineLearning #BigData…
##  2     2 Good list: Frequently updated and active #MachineLearning blogs http://t.co/4rw1alb6Zz http://t.co/3yr6o0lIUg
##  3     3 Information Designer for Facebook Timeline, on Data #Visualization and #BigData http://t.co/VkQoOLK3sE http:/…
##  4     4 Top /r/MachineLearning posts, Jan http://t.co/fWgZJLB5qJ
##  5     5 Free #MachineLearning and Predictive #Analytics Training with Microsoft Virtual Academy on #AzureML http://t.…
##  6     6 #Data Scientists most happy doing #Predictive Analysis, least when data cleaning @CrowdFlower http://t.co/1ll…
##  7     7 Cartoon: Data Scientist gets 3 wishes for Valentine's Day http://t.co/yFOPmSTazV
##  8     8 SUTD: Postdoctoral Fellowship at MIT and SUTD http://t.co/zwifLjpbYm
##  9     9 Localytics: Data Scientist http://t.co/oKhgVFtfBP
## 10    10 My Brief Guide to Big Data and Predictive Analytics for non-experts http://t.co/ouQEGEqFfq
## # ... with 390 more rows
```

```
(tweets <- tweets %>% mutate(hashtag = str_extract_all(tweet, "#(\\w)+"), tag_no = str_count(tweet, "#(\\w)+"),
                  url = str_extract_all(tweet, "http://t.co/(\\w)*"),
                  url_no = str_count(tweet, "http://t.co/(\\w)*")))
```

```
## # A tibble: 400 x 6
##    index tweet                                                                           hashtag  tag_no url     url_no
##    <int> <chr>                                                                           <list>    <int> <list>   <int>
##  1     1 Metaps, Japanese App Monetization firm, raises $36M series C to push Into #Big… <chr [3…      3 <chr […      1
##  2     2 Good list: Frequently updated and active #MachineLearning blogs http://t.co/4r… <chr [1…      1 <chr […      2
##  3     3 Information Designer for Facebook Timeline, on Data #Visualization and #BigDat… <chr [2…      2 <chr […      2
##  4     4 Top /r/MachineLearning posts, Jan http://t.co/fWgZJLB5qJ                        <chr [0…      0 <chr […      1
##  5     5 Free #MachineLearning and Predictive #Analytics Training with Microsoft Virtua… <chr [3…      3 <chr […      1
##  6     6 #Data Scientists most happy doing #Predictive Analysis, least when data cleani… <chr [2…      2 <chr […      2
##  7     7 Cartoon: Data Scientist gets 3 wishes for Valentine's Day http://t.co/yFOPmSTa… <chr [0…      0 <chr […      1
##  8     8 SUTD: Postdoctoral Fellowship at MIT and SUTD http://t.co/zwifLjpbYm            <chr [0…      0 <chr […      1
##  9     9 Localytics: Data Scientist http://t.co/oKhgVFtfBP                               <chr [0…      0 <chr […      1
## 10    10 My Brief Guide to Big Data and Predictive Analytics for non-experts http://t.c… <chr [0…      0 <chr […      1
## # ... with 390 more rows
```

`unnest` in `tidyr`

When we have a list-column, `unnest` makes each element of the list its own row.

```
str(unnest)
```

```
## function (data, ..., .drop = NA, .id = NULL, .sep = NULL, .preserve = NULL)
```

```
tweets %>% unnest(hashtag) %>% mutate(tweet = NULL)
```

```
## # A tibble: 296 x 4
##    index tag_no url_no hashtag
##    <int>  <int>  <int> <chr>
##  1     1      3      1 #BigData
##  2     1      3      1 #MachineLearning
##  3     1      3      1 #BigDataCo
##  4     2      1      2 #MachineLearning
##  5     3      2      2 #Visualization
##  6     3      2      2 #BigData
##  7     5      3      1 #MachineLearning
##  8     5      3      1 #Analytics
##  9     5      3      1 #AzureML
## 10     6      2      2 #Data
## # ... with 286 more rows
```

```
tweets %>% unnest(url) %>% mutate(tweet = NULL)
```

```
## # A tibble: 450 x 4
##    index tag_no url_no url
##    <int>  <int>  <int> <chr>
##  1     1      3      1 http://t.co/ly3sS2fpdb
##  2     2      1      2 http://t.co/4rw1alb6Zz
##  3     2      1      2 http://t.co/3yr6o0lIUg
##  4     3      2      2 http://t.co/VkQoOLK3sE
##  5     3      2      2 http://t.co/LoEZ3DL3FX
##  6     4      0      1 http://t.co/fWgZJLB5qJ
##  7     5      3      1 http://t.co/zGCVYYQbgr
##  8     6      2      2 http://t.co/lllrqztvZc
##  9     6      2      2 http://t.co/qQ7YbaV9Rz
## 10     7      0      1 http://t.co/yFOPmSTazV
## # ... with 440 more rows
```

extract in tidyr

Given a regular expression with capturing groups, `extract()` turns each group into a new column:

```
str(extract)
```

```
## function (data, col, into, regex = "([[:alnum:]]+)", remove = TRUE, convert = FALSE, ...)
```

```
df <- data.frame(x = c(NA, "a-b", "a-d", "b-c", "d-e"))
df %>% extract(x, c("A", "B"), "([a-d]+)-([a-d]+)")  # If no match, NA
```

```
##      A    B
## 1 <NA> <NA>
## 2    a    b
## 3    a    d
## 4    b    c
## 5 <NA> <NA>
```

Locale Sensitive Operations

Use `tolower()` or `toupper()` to do case folding/conversion in base R:

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

```
## [1] "business programming in r"
```

However, different languages have different rules for changing case, e.g.,:

```
# Turkish has two i's: with and without a dot, and it has a different rule for capitalising them
toupper(c("i", "ı"))
```

```
## [1] "I" "I"
```

We can use `str_to_upper()` to pick which set of rules to use by specifying a locale with an ISO 639 language code:

```
str_to_upper(c("i", "ı"), locale = "tr")  # If left blank, use the current locale provided by the OS.
```

```
## [1] "İ" "I"
```

Wikipedia has a good list of the codes.

Locales also affect sorting. `order()` and `sort()` in base R sort strings using the current locale.

To have robust behaviour across different computers, use `str_sort()` and `str_order()` which take an additional `locale` argument:

```
str_sort(c("apple", "eggplant", "banana"), locale = "en")  # English
```

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

```
str_sort(c("apple", "eggplant", "banana"), locale = "haw")  # Hawaiian
```

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