Basic String Manipulation in R

36-600

Fall 2021

What is a String?

A string is a sequence of characters that are bound together, where a character is a symbol is a written language.

In `R`, a string is of class `character` and is bounded by quotes (either single or double). Double quotes are preferable, because then one can use single quotes as apostrophes in strings.

In this set of notes, we will focus on common functions that one might use to analyze string-based data.

For our example data, we will look at the text of Hillary Clinton's acceptance of the Democratic Party nomination in 2016 (which I just happen to have lying around from its use in another class). We will use the `readLines()` function, which reads in the data line-by-line.

```
lines <- readLines("http://www.stat.cmu.edu/~pfreeman/clinton.txt")
lines[1]
```

```
## [1] "Thank you all very, very much! Thank you for that amazing welcome!"
```

Basic String Functions

To concatenate strings, use the `paste()` function:

```
paste(lines[1],lines[2])
```

```
## [1] "Thank you all very, very much! Thank you for that amazing welcome! Thank you all for the great convention tha
```

Note that by default, the strings are concatenated with a space separating them. This is controlled by the `sep` argument:

```
paste(lines[1],lines[2],sep=" 1 2 3 ")
```

```
## [1] "Thank you all very, very much! Thank you for that amazing welcome! 1 2 3 Thank you all for the great conventi
```

Note that above, we pasted two separate arguments together. We can also pass in a vector of strings, in which case the separation is controlled by the `collapse` argument:

```
paste(lines[1:2],collapse=" ")
```

```
## [1] "Thank you all very, very much! Thank you for that amazing welcome! Thank you all for the great convention tha
```

`sep` and `collapse` can be combined:

```
paste(c("a","b"),c("1","2"),sep="+") ; paste(c("a","b"),c("1","2"),sep="+",collapse=",")
```

```
## [1] "a+1" "b+2"
```

```
## [1] "a+1,b+2"
```

Basic String Functions

To count the number of characters in a string, use the function `nchar()` (as opposed to `length()`, which counts the number of elements in a vector).

To illustrate the difference:

```
length(lines)       # number of lines input via readLines()
```

```
## [1] 154
```

```
nchar(lines)[1:3]   # number of characters in each of the first three lines
```

```
## [1]  66  54 180
```

If you, for instance, want to extract the first and last ten characters in one of the lines, you would use `substr()`:

```
substr(lines[54],1,10)                                # characters 1 through 10 of line 54
```

```
## [1] "So we gath"
```

```
substr(lines[54],nchar(lines[54])-9,nchar(lines[54])) # ...and the last ten characters
```

```
## [1] "to school."
```

String Splitting

Suppose you want to split a string on a particular character (the most common example of which being splitting on spaces, to get all the words in a string). To do this, you would apply the `strsplit()` function:

```
strsplit(lines[1:2],split=" ")  # split each of the first two lines on spaces
```

```
## [[1]]
##  [1] "Thank"    "you"      "all"      "very,"    "very"     "much!"    "Thank"    "you"      "for"      "that"
## [12] "welcome!"
##
## [[2]]
##  [1] "Thank"      "you"        "all"        "for"        "the"        "great"      "convention" "that"       "we'v
## [10] "had."
```

We immediately note the following.

- `strsplit()` returns a list, with each element of the list mapping back to each element of the string vector that was input. Here, we input two lines, and got back a list with two elements.
- Each list element contains a vector of split-up characters.
- Not every output string is actually a word. We have to do something more complicated to get a better word list.

String Splitting and Regular Expressions

Regular expressions, or regexes, define a rabbit hole that extends deep into the Earth. They are specially constructed strings that allow for flexible pattern matching. The rules for constructing regexes are independent of `R`; you may already know them.

We will focus right now on the use of square brackets and metacharacters to define a regex.

Square brackets: we want to match any one character that appears inside.

- "[abcde]" means "look for any string that contains a, b, c, d, or e" (case sensitive!)
- "[a-e]" means the same thing; the dash denotes a range
- "[^a-e]" means "look for any string that contain characters other than a, b, c, d, or e"
- " [1-4][2-6] " matches strings that contain the numbers 12-16, 22-26, 32-36, or 42-46

For our immediate purposes, let's split the line from the speech on spaces *and* commas:

```
strsplit(lines[1:2],split="[ ,]") # not space, then comma, but space or comma
```

```
## [[1]]
##  [1] "Thank"    "you"      "all"      "very"     ""         "very"     "much!"    "Thank"    "you"      "for"
## [12] "amazing"  "welcome!"
##
## [[2]]
##  [1] "Thank"      "you"        "all"        "for"        "the"        "great"      "convention" "that"       "we'v
## [10] "had."
```

There is still work to do.

String Splitting and Regular Expressions

Commonly used metacharacters include

- "[[:alnum:]]" is the same as "[a-zA-Z0-9]"
- "[[:punct:]]" means "match any string that contains a punctuation mark"
- "[[:space:]]" means "match any string that contains a space, a tab, or a new line"

```r
strsplit(lines[1:2],split="( |[[:punct:]])") # split on space or a punctuation mark
```

```
## [[1]]
##  [1] "Thank"   "you"     "all"     "very"    ""        "very"    "much"    ""        "Thank"   "you"     "for"
## [13] "amazing" "welcome"
## 
## [[2]]
##  [1] "Thank"      "you"        "all"        "for"        "the"        "great"      "convention" "that"       "we"
## [10] "ve"         "had"
```

But we've split on the apostrophe (and we have empty strings). We'd have to go back and split on specific punctuation marks. But we'll stop here because we've basically made our point...

String Splitting and Regular Expressions

…well, except for one last thing. In `R`, the characters

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

are all special characters. To find occurrences of these symbols in strings, we use an *escape sequence*: we place a backslash in front of the symbol. But there's an issue with that: given that the backslash is a special character, it itself needs to be escaped. See below. (Sigh.)

```
strsplit(lines[1:2],split="[ !\\.]") # split on spaces, exclamation points, and escaped periods
```

```
## [[1]]
##  [1] "Thank"   "you"     "all"     "very,"   "very"    "much"    ""        "Thank"   "you"     "for"     "that"
## [13] "welcome"
##
## [[2]]
##  [1] "Thank"      "you"        "all"        "for"        "the"        "great"      "convention" "that"       "we'v
## [10] "had"
```

Word Tables

What are the 20 most common words in Clinton's speech? Let's get to it!

```
sort(table(unlist(strsplit(lines,split="[ !\\.]"))),decreasing=TRUE)[1:20]
```

```
## 
##    the            to   and     a    of    in   our    we   you   And   for     I  that    is    it pe
##    200   180   176   160   102    98    77    70    69    68    57    57    57    57    41    37
##   with   are 
##     36    35 
```

We note a few more things here:

- `unlist()` is a way to concatenate all the elements of a list together into a single vector;
- we need to do some additional processing to remove the empty strings (second place with 180, generally from the ends of lines); and
- case sensitivity impacts the count (e.g., "and" and "And" were treated separately)...this can be mitigated by applying the `tolower()` function after `unlist()`, which converts all letters to lower case.

String Searching

If you want to see if a particular word appears in a line, you can use the `grep` family of functions. For instance, if you want to determine if "And" occurs on a line, use `grepl()`, which turns `TRUE` or `FALSE`:

```
grepl("And",lines[1:5])
```

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

String searches allow for regexes:

```
grepl("(and|And)",lines[1:5])
```

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

`grep()` itself either returns the number of the line in which the string is observed (here, lines 3 and 5)

```
grep("and",lines[1:5])
```

```
## [1] 3 5
```

or, if you pass in the argument `value=TRUE`, you get the lines themselves

```
grep("and",lines[1:5],value=TRUE)
```

```
## [1] "And Chelsea, thank you. I am so proud to be your mother and so proud of the woman you've become. Thank you fo
## [2] "You know, that conversation has lasted through good times that filled us with joy and hard times that tested
```

Dynamic String Extraction

We saw above that we can use `substr()` to extract a substring. However, we need to specify where the substring starts and where it ends. A more dynamic extractor involves combining `gregexpr()` and `regmatches()`. Here, let's extract every occurrence of "and" and "And" in the first five lines:

```
out     <- gregexpr("(a|A)nd",lines[1:5])
matches <- regmatches(lines[1:5],out)     # outputs a list, one for each line
unlist(matches)
```

```
## [1] "And" "and" "and" "and" "And" "and" "And"
```

Removing/Replacing Characters

Let's say that instead of splitting on punctuation, as we (eventually) did above, we want to remove or replace them. One way to do that is to use `gsub()`:

```
gsub("[[:punct:]]","-",lines[1])
```

```
## [1] "Thank you all very- very much- Thank you for that amazing welcome-"
```

Stopwords

One last thing to look at is the removal of uninformative *stopwords* from a document. (It is often the case that we'd want to remove stopwords like "a", "and", and "the," etc., before doing any real statistical analysis.)

```
library(stopwords)
head(stopwords("en"),10)
```

```
##  [1] "i"          "me"         "my"         "myself"     "we"         "our"        "ours"       "ourselves"  "you"        "
```

Here's what Clinton's speech looks like after stopword removal. It's messy and imperfect, but it's a start.

```
speech           <- tolower(unlist(strsplit(lines,split="[ ,!\\.]")))
w                <- which(nchar(speech)==0)
speech           <- speech[-w]  # could do speech <- speech[speech!=""] also, or dplyr...
stopword.logical <- speech %in% stopwords("en")      # is element of left "in" vector at right? [T/F]
paste(speech[stopword.logical==FALSE],collapse=" ")
```

```
## [1] "thank much thank amazing welcome thank great convention chelsea thank proud mother proud woman become thank b
```