The Basics of Working with Strings

Today, we'll continue our exploration of working with different types of data by working with strings. Strings are textual data: the make and model of a car in our `mpg` data, or the airline, origin, and destination in our `flights` dataset. Working with strings is incredibly useful for analyzing data that's based on text, like news articles, tweets, and more! We'll learn the basics of working with strings today, and Chapter 14 of the textbook has way more on them. This is a brief introduction to them so that in later courses we can do more with tools like web scraping.

At the outset, please note that because of how R outputs the results of string searches, this document looks best as an HTML file (for me, it displays best in Chrome). We have provided a Word version simply in case you want to print it, but please note that it will be missing much of the output, so to see the results, you'll need to look at the HTML file or be entering in the code yourself.

In this first handout, we'll cover some basics of how to work with strings and find patterns in strings. In future classes (DATA 301 and DATA 401), we'll delve more deeply into manipulating strings with regular expressions, which really opens up a whole new world of possibilities!

Whenever you write a string in R, you enclose it in quotation marks. So, for example, this is a string:

```
## Make a string
x <- "This is a string I made for DATA 101"
y <- "To include  \"quotation marks\" in my string, I need to use a
backslash."
x
```

```
## [1] "This is a string I made for DATA 101"
```

```
y
```

```
## [1] "To include  \"quotation marks\" in my string, I need to use a
backslash."
```

```
writeLines(y)
```

```
## To include  "quotation marks" in my string, I need to use a backslash.
```

```
typeof(x)
```

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

```
typeof(y)
```

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

There are three things to note here:

1. In R, the quotation marks are how you tell R you are using a string. So if I want to include a string that includes quotation marks, I need to use a backslash (\) to tell R that I'm not starting/ending a string, I'm just trying to include quotation marks in my string. There are a set of special characters (described below) which operate this way.

2. When I display the raw string `y`, it shows up with the `\"` (the backslash plus quote). If I just want to see it without this, I need to use the `writeLines()` function.

3. Note that when I use the `typeof()` function, R tells us that `x` and `y` are characters, which is just R-speak for strings.

But quotation marks are not the only special characters in R. There is a set of special characters in strings that need to be preceded by a backslash. The most common ones are:

- `\n`: new character
- `\'`: single quote
- `\"`: double quote
- `\\`: backslash

To see all of them, type `?'"'` in your R console to bring up the help file for strings.

The key set of functions today are in the `stringr` library, which is part of the tidyverse. Note that `stringr` is another fun name: `stringr` is how you use strings in R! To work with some of the tools of `stringr`, we also need the `htmlwidgets` package, which we'll load now as well.

```
library(tidyverse)
library(htmlwidgets)
```

The most common set of string tools are the ones used to count the length of string, and to combine and split (subset) strings. For example, I can use `str_length()` to see how long a string is, and I can use `str_c()` to combine strings:

```
str_length(x)
```

```
## [1] 36
```

```
str_c(x,y)
```

```
## [1] "This is a string I made for DATA 101To include  \"quotation marks\"
in my string, I need to use a backslash."
```

Note that when I use `str_c()`, it just combines the strings with no spaces, so that's not very nice looking. I can use the `sep` option to say how they should be separated:

```
str_c(x,y, sep=". ")
```

```
## [1] "This is a string I made for DATA 101. To include  \"quotation marks\"
in my string, I need to use a backslash."
```

So here, by adding `sep = ". "`, I add a period (since they're separate sentences), and I get a space between them, making them much easier to read. If you want to collapse two strings into one, just use the `collapse` option:

```
z <- str_c(c(x,y),collapse=". ")
z

## [1] "This is a string I made for DATA 101. To include  \"quotation marks\"
in my string, I need to use a backslash."
```

You use the `str_sub()` to subset a portion of a string. To get the first 5 characters of a string, I would write:

```
str_sub(x,1,5)

## [1] "This "

str_sub(y,1,5)

## [1] "To in"
```

So the `str_sub()` function takes a string, and then extracts from the beginning to the end of a character sequence. To get the 3rd through the 6th characters of x, I would write:

```
str_sub(x,3,6)

## [1] "is i"
```

Note here that spaces are included as characters. The `str_sub()` is the most common way of extracting information from a string. If you just wanted to extract a word from a string, you can use the `word()` function. For example, to extract the second word of x ("is"), we would write:

```
word(x,2)

## [1] "is"
```

There are a few other useful string functions you can look up on your own:

- `str_to_lower`: converts a string to all lower-case letters
- `str_to_upper`: converts a string to all upper-case letters
- `str_trim`: trims white space from a string
- `str_pad`: adds white space to a string
- `str_sort`: sorts strings lexically (i.e., in alphabetical order)

Really, there are a ton more. If you try to learn all of these functions, you'll drive yourself crazy. Instead, what I'd do is learn a few of the basics, and as you encounter problems, do some googling and find the one that you need for that issue.

The (Very) Basics of Pattern Matching and Regular Expressions

Regular expressions (regexps) are a powerful tool for working with textual data. Basically, regular expression match patterns in strings. We'll use the `str_view()` function to do this.

Let's start with a simple list of words, and match patterns in those words. Again, we'll keep this simple, and then see a more interesting example below. Let's start with the easiest case: an exact match.

We'll take a list of fruits and ask it to tell us which fruits have the letters "an" in them:

```
## Pattern matching
word_list <- c("apple", "banana", "grape","orange","pear","pineapple")
## Exact pattern matching
str_view(word_list, "an")
```

So we see that both "banana" and "orange" have the string "an" in them.

We can generalize this a bit, by searching for `.`, which matches any character. For example, let's find any instance of "a" not at the beginning or ending of a word:

```
str_view(word_list, ".a.")

str_view(word_list, ".a.", match=T)
```

So note that "a" occurs in the middle of every fruit except "apple". While apple obviously contains the letter a, it comes at the beginning, and so the `.a.` pattern is not matched! Note that the `match=T` option just tells R to only return the cases where you find a match. This is helpful when you have a long list.

So if `.` matches any character, how do you match a period? You need to treat it as a special character! So, for example:

```
str_view(c("abc","a.b.c","a.c","def"), "a\\.b")
```

By default, a regular expression matches any part of the string. But often, we might want to find words that begin or end with a certain set of letters. We can do that with the anchors:

- `^` (shift + 6) matches the start of a string
- `$` (shift + 4) matches the end of a string

So let's see this in action:

```
## anchors
str_view(word_list, "^a")

str_view(word_list,"e$")
```

So while only "apple" begins with a, several fruits end in e: apple, grape, orange, and pineapple. So note that if I want to force matches only with a complete string, I anchor with both ^ and $:

```
## using both anchors
str_view(word_list, "apple")

str_view(word_list, "^apple$")
```

Test Yourself: Using the words Dataset

In the stringr library, there is a dataset of words (helpfully, called `words`) that is included for practice with regular expressions. Using the tools we just reviewed, find all words that:

- Begin with the letter "k"
- End with the letter "s"
- Have at least 10 letters
- Are exactly 11 letters long

For the last two challenges, try not to use the `str_length()` functions.

Answer:

```
## Test yourself
str_view(words, "^k", match=T)

str_view(words, "x$", match=T)

str_view(words, ".........", match=T)

str_view(words, "^...........$", match=T)
```

For the length items, note that we can take advantage of the fact that `.` matches any character.

You can also generalize this matching a bit with a set of commands. We already saw that `.` matches any character. There are a few other useful symbols you can use for generalized matching:

- `\d` matches any digit (0, 1, 2, etc.)
- `\s` matches any white space (i.e., space, tab, or new line)
- `[abcd]` matches a, b, c, or d
- `[^abcd]` matches anything *except* a, b, c, or d

Let's see this in action. Let's find all words in the `words` dataset that begin with a vowel:

```
str_view(words, "^[aeiou]", match=T)
```

I've suppressed the list because it is extremely long. But notice the syntax: I'm using the `^` to mark the beginning of the word, and then `[aeiou]` to say that the letter should be an a, e, i, o, or u (i.e., a vowel).

Test Yourself: Words beginning with consonants

How would you modify the syntax above to list the words in `words` that begin with consonants?

Answer:

```
str_view(words, "^[^aeiou]", match=T)
```

Here, I'm suppressing the output since the list is very long, but notice that I'm using the fact that `[^aeiou]` matches anything *except* a vowel. So I could also have inputted the list of consonants, but this is an easier shorthand to doing that.

Test Yourself: Word Patterns

When you were in school, you probably learned the simple rule: "i before e, except after c". Let's see if this rule holds! We'll proceed in a few steps.

1. Are there examples of the sequence "cie"? This breaks the rule, since it should be "cei"
2. Can we find words where we see the sequence "ei" not following a c?

Answer:

```
## i before e?
str_view(words, "cie", match=T)

str_view(words, "[^c]ei", match=T)
```

Note here that the `[^c]ei` tells R that we want to find the string "ei" after any letter but `c`. If we'd typed `^cei` it would have searched for words starting with `cei`, which is not what we wanted.

Notice that in the results, we do see a few examples of words that break the pattern! As usual, "rules" in English are more suggestions than actual rules.

In R, when you use the `[]`, you are creating a character class. We saw this above: for example, `[abc]` matches a, b, or c. But there are some more powerful classes you can use as well:

- `[a-z]` matches any lower-case character between a and z (i.e., the English alphabet)
- `[A-Z]` matches any upper-case character between a and z
- `[:alpha:]`: matches any letter
- `[:lower:]`: matches any lower-case letter
- `[:upper:]`: matches any upper-case letter
- `[:digit:]`: matches any digit
- `[:alnum:]`: matches any letter or number

There are more options available, see the help file for more details. We'll see how to use these below.

You can also alternate to pick between patterns with the | symbol; remember we've used this before as "or". For example, I could find the words that end in "-ing" or "-ise":

```
## using the "|" operator
str_view(words, "i(ng|se)$", match=T)
```

Test Yourself: Word Beginnings

Using the `words` dataset, find words that begin with "ab" or "ad"

Answer:

```
str_view(words, "^ab|^ad", match=T)
```

Patterns and Repetition

We can also control how many times a pattern matches:

- `?`: 0 or 1 times
- `+`: 1 or more times
- `*`: 0 or more times

For example, let's see if the pattern "cc" appears in any words in the `words` dataset:

```
str_view(words, "cc+", match=T)
```

But rather than `?`, `+`, and `*`, I prefer to specify the exact number of matches:

- `{n}`: exactly `n` matches
- `{n,}`: `n` or more matches
- `{,m}`: at most `m` matches
- `{n,m}`: between n and m matches

So we could replicate the code above as:

```
str_view(words, "c{2,}", match=T)
```

Test Yourself: More fun with words

Using the words dataset, find:

1. The words that contain only consonants
2. Words that begin with 3 or more consonants

```
str_view(words, "^[^aeiou]{1,}$", match=T)

str_view(words,"^[^aeiou]{3,}", match=T)
```

You can also use parentheses to create capturing groups to find patterns in words. These become useful because you can refer to them with a backreference such as `\\1`. For example, the following syntax finds repeated pairs of letters:

```
str_view(words, "(..)\\1", match=T)
```

So here, "em" is repeated in "remember." Let's break down that syntax. We know what `.` does: it matches any character. So `(..)` creates a capturing group of any two letters, call it group 1. The `\\1` just says match group 1 to itself (i.e., find pairs of repeated letters). So this says find me any pair of letters that are repeated in any word.

This seems confusing, but just takes practice. For example, if I wanted to find examples of repeated letters, I could just write:

```
str_view(words, "(.)\\1", match=T)
```

I've suppressed the output, because it's a long list, but I just told R to find any character with `.`, create a group out of it `(.)`, and then find it again (`\\1`): in short, repeated letters!

I could also find words that begin and end with the same letter:

```
str_view(words, "^(.).{0,}\\1$", match=T)
```

Again, let's break down the syntax. We tell R to find words that begin with any letter `^(.)`. The middle part just says then have any number of characters: `.{0,}` means have any character 0 or more times (to handle words of any length), and then end with our original character: `\\1$`.

Test Yourself: A letter 3 or more times

How would you find words that contain the same letter at least 3 times? For example, the word "eleven" has 3 "e"s.

```
str_view(words, "(.).{0,}\\1.{0,}\\1", match=T)
```

This is just an expansion on the "begins and ends with the same letter" syntax from above.

String Functions

Now we can use regular expressions to do more than just give us a list of words (not that it's not useful!). Now let's see how to detect strings for now, and then in future courses in this sequence (DATA 301 and 401), we'll explore regular expressions in more detail to truly explore their power.

Let's begin by detecting a match, for which we use the `str_detect()` function. This function returns a logical vector that is `True` if the value is matched, and `False` if the value is not. For example:

```
str_detect(word_list,"e")

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

So this is telling me that the letter e appears in apple, grape, orange, pear, and pineapple, but not in banana. Typically, we'll use this with `sum()` or `mean()` to give us counts or percentages of matches. For example, how many words in the words database start with the letter p? What percentage end in a vowel?

```
sum(str_detect(words,"^p"))
```

```
## [1] 72
```

```
mean(str_detect(words, "[aeiou]$"))
```

```
## [1] 0.2765306
```

Remember, if I have a logical factor (True/False), then sum gives me the total number of values where it is True, and mean gives me the percentage of cases where it is True. So here, 72 words begin with the letter p, and 27.65% of words end in a vowel (mostly "e").

Test Yourself: How many words do not have vowels?

Using the `words` dataset, find out how many words have no vowels.

```
sum(str_detect(words, "^[^aeiou]{1,}$"))
```

```
## [1] 6
```

Note that this is a slight variation on a call we made above where we listed these word when we found the percentage of words that ended in a vowel.

You can also use the related function `str_subset()` which gives you the subset of words that match a given string. For example, if I wanted to find the subset of words that end in x, I could write:

```
str_subset(words, "x$")
```

```
## [1] "box" "sex" "six" "tax"
```

So far, we've been working with `words` as it comes to us: a long vector (just a series of words). But let's convert it to a tibble (data frame) so that we can use our `dplyr` functions like mutate and filter:

```
df <- tibble(
  word = words,
  i = seq_along(word))
df %>%
  filter(str_detect(word, "x$"))
```

```
## # A tibble: 4 x 2
##   word      i
##   <chr> <int>
## 1 box     108
## 2 sex     747
## 3 six     772
## 4 tax     841
```

So that just replicates what we had above, but in a slightly more realistic setting (the `i` variable just tells us what order the word appeared in our list).

There's also a variation on `str_detect()` called `str_count()` that counts the number of matches in a word. So let's now count the number of vowels and consonants in each word:

```
df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]"))
```

```
## # A tibble: 980 x 4
##    word          i vowels consonants
##    <chr>     <int>  <int>      <int>
##  1 a             1      1          0
##  2 able          2      2          2
##  3 about         3      3          2
##  4 absolute      4      4          4
##  5 accept        5      2          4
##  6 account       6      3          4
##  7 achieve       7      4          3
##  8 across        8      2          4
##  9 act           9      1          2
## 10 active       10      3          3
## # … with 970 more rows
```

So, for example, the word "able" has two vowels (a,e) and two consonants (b,l).

Test Yourself: Some Fun with string detection and subsetting

Using these functions, find:

- All words that start with a vowel and end with a consonant

- Find the words with the highest number of vowels

```
## words starting with vowels \& ending with a consonant
df %>%
  filter(str_detect(words, "^[aeiou].{0,}[^aeiou]$"))
```

```
## # A tibble: 122 x 2
##    word        i
##    <chr>   <int>
##  1 about       3
##  2 accept      5
##  3 account     6
##  4 across      8
##  5 act         9
##  6 actual     11
##  7 add        12
##  8 address    13
##  9 admit      14
## 10 affect     16
## # … with 112 more rows
```

```
## words with the highest number of vowels
df %>%
  mutate(nvowel = str_count(word,"[aeiou]")) %>%
  arrange(desc(nvowel))
```

```
## # A tibble: 980 x 3
##    word            i nvowel
##    <chr>       <int>  <int>
##  1 appropriate    48      5
##  2 associate      57      5
##  3 available      62      5
##  4 colleague     166      5
##  5 encourage     268      5
##  6 experience    292      5
##  7 individual    423      5
##  8 television    846      5
##  9 absolute        4      4
## 10 achieve         7      4
## # … with 970 more rows
```

Appendix: All code used to make this document

```
knitr::opts_chunk$set(echo = TRUE)
knitr::opts_knit$set(root.dir = '~/Dropbox/DATA101')
library(tidyverse)
## Make a string
x <- "This is a string I made for DATA 101"
y <- "To include  \"quotation marks\" in my string, I need to use a
backslash."
x
y
writeLines(y)
typeof(x)
typeof(y)
library(tidyverse)
library(htmlwidgets)
str_length(x)
str_c(x,y)
str_c(x,y, sep=". ")
z <- str_c(c(x,y),collapse=". ")
z
str_sub(x,1,5)
str_sub(y,1,5)
str_sub(x,3,6)
word(x,2)
## Pattern matching
word_list <- c("apple", "banana", "grape","orange","pear","pineapple")
## Exact pattern matching
str_view(word_list, "an")
str_view(word_list, ".a.")
str_view(word_list, ".a.", match=T)
```

```r
str_view(c("abc","a.b.c","a.c","def"), "a\\.b")
## anchors
str_view(word_list, "^a")
str_view(word_list,"e$")
## using both anchors
str_view(word_list, "apple")
str_view(word_list, "^apple$")
## Test yourself
str_view(words, "^k", match=T)
str_view(words, "x$", match=T)
str_view(words, ".........", match=T)
str_view(words, "^.........$", match=T)
str_view(words, "^[aeiou]", match=T)
str_view(words, "^[^aeiou]", match=T)
## i before e?
str_view(words, "cie", match=T)
str_view(words, "[^c]ei", match=T)
## using the "|" operator
str_view(words, "i(ng|se)$", match=T)
str_view(words, "^ab|^ad", match=T)
str_view(words, "cc+", match=T)
str_view(words, "c{2,}", match=T)
str_view(words, "^[^aeiou]{1,}$", match=T)
str_view(words,"^[^aeiou]{3,}", match=T)
str_view(words, "(..)\\1", match=T)
str_view(words, "(.)\\1", match=T)
str_view(words, "^(.).{0,}\\1$", match=T)
str_view(words, "(.).{0,}\\1.{0,}\\1", match=T)
str_detect(word_list,"e")
sum(str_detect(words,"^p"))
mean(str_detect(words, "[aeiou]$"))
sum(str_detect(words, "^[^aeiou]{1,}$"))
str_subset(words, "x$")
df <- tibble(
  word = words,
  i = seq_along(word))
df %>%
  filter(str_detect(word, "x$"))
df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]"))
## words starting with vowels \& ending with a consonant
df %>%
  filter(str_detect(words, "^[aeiou].{0,}[^aeiou]$"))
## words with the highest number of vowels
df %>%
  mutate(nvowel = str_count(word,"[aeiou]")) %>%
  arrange(desc(nvowel))
```