Assignment 3

Zachary Palmore

9/11/2020

Introduction

There are four main topics for this week. They include:

- Identify and view strings from a list of strings
- Transform and format messy strings
- Describe expressions of strings
- Construct expressions of strings

As you can see, each of them will use strings that we generate as well.

For these topics we will use the tidyverse package which contains all that is necessary for our purposes.

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------------------------------------------- ti

## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.2
## v tidyr   1.1.1     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0

## -- Conflicts ------------------------------------------------------------------------------ tidyvers
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

The Data Data is loaded from FiveThirtyEight's Github repository.

```r
# Pulling in from the source on Github
majors <- read.csv(
  "https://raw.githubusercontent.com/fivethirtyeight/data/master/college-majors/majors-list.csv")
glimpse(majors)
```

```
## Rows: 174
## Columns: 3
## $ FOD1P          <fct> 1100, 1101, 1102, 1103, 1104, 1105, 1106, 1199, 1302...
## $ Major          <fct> GENERAL AGRICULTURE, AGRICULTURE PRODUCTION AND MANA...
## $ Major_Category <fct> Agriculture & Natural Resources, Agriculture & Natur...
```

The variables are FOD1P, Major, and Major_Category with 174 rows.

Majors in Data and Statistics

Identifying the majors that contain either DATA or STATISTICS could also mean identifying majors that contain both. For this reason, both are pulled individually using a few different options. The function that will be used in each of them is "grep".

Option 1 Find the index values where the major's have statistics and/or data in the character strings.

```r
# This method finds the index values
# where statistics and data are present in the strings
grep(pattern="Statistics", majors$Major, ignore.case = TRUE)
```

```
## [1] 44 59
```

```r
grep(pattern="Data", majors$Major, ignore.case = TRUE)
```

```
## [1] 52
```

Then extract those strings using the index values as a reference.

```r
# This extracts those index values
majors$Major[44]
```

```
## [1] MANAGEMENT INFORMATION SYSTEMS AND STATISTICS
## 174 Levels: ACCOUNTING ACTUARIAL SCIENCE ... ZOOLOGY
```

```r
majors$Major[52]
```

```
## [1] COMPUTER PROGRAMMING AND DATA PROCESSING
## 174 Levels: ACCOUNTING ACTUARIAL SCIENCE ... ZOOLOGY
```

```r
majors$Major[59]
```

```
## [1] STATISTICS AND DECISION SCIENCE
## 174 Levels: ACCOUNTING ACTUARIAL SCIENCE ... ZOOLOGY
```

Option 2 Alternatively, one could also select the values in the function with the value = TRUE parameter to display the strings that meet the criteria of containing the letters "data" or "statistics" without matching case.

```r
# Alternatively, grep also allows you to select the values too
grep(pattern="Statistics", majors$Major, ignore.case = TRUE, value = TRUE)
```

```
## [1] "MANAGEMENT INFORMATION SYSTEMS AND STATISTICS"
## [2] "STATISTICS AND DECISION SCIENCE"
```

```r
grep(pattern = "Data", majors$Major, ignore.case = TRUE, value = TRUE)
```

```
## [1] "COMPUTER PROGRAMMING AND DATA PROCESSING"
```

Option 3 The same function can also be shortened. Here we ask grep to find where the pattern contains both statistics and data using '|' to indicate 'and'.

```r
grep(pattern = "Data|Statistics", majors$Major, ignore.case = TRUE, value = TRUE)
```

```
## [1] "MANAGEMENT INFORMATION SYSTEMS AND STATISTICS"
## [2] "COMPUTER PROGRAMMING AND DATA PROCESSING"
## [3] "STATISTICS AND DECISION SCIENCE"
```

There are likely plenty of options without using grep. However, the results should be the same. Those majors that contain either DATA or STATISTICS are:

- MANAGEMENT INFORMATION SYSTEMS AND STATISTICS
- COMPUTER PROGRAMMING AND DATA PROCESSING
- STATISTICS AND DECISION SCIENCE

I wonder; how many with data (or data|statistics separately) would fall into the math Major_Category?

Tranforming Data

It seems the challenge is to transform data that looks like this,

```r
Fruits <- '[1] "bell pepper"  "bilberry"     "blackberry" "blood orange"
[5] "blueberry"    "cantaloupe"   "chili pepper" "cloudberry"
[9] "elderberry"   "lime"         "lychee"       "mulberry"
[13] "olive"        "salal berry"'
Fruits
```

```
## [1] "[1] \"bell pepper\"  \"bilberry\"     \"blackberry\" \"blood orange\"\n[5] \"blueberry\"    \"ca
```

into a format that is better suited for manipulating the data. The example solution to strive for is this:

```r
Fruit_example <- c("bell pepper", "bilberry", "blackberry",
                   "blood orange", "blueberry", "cantaloupe",
                   "chili pepper", "cloudberry", "elderberry",
                   "lime", "lychee", "mulberry", "olive",
                   "salal berry")
```

The *fruits* list displays a disgustingly formatted list of fruits. It is also not usable for any querying. To clean this up, and produce a list of these same values, we can split the string of fruits by removing their common separator "\". It appears each fruit that we want to see in our final list has a "\" before and after it.

```
# Creating a empty list to pump data into
EmptyList <- vector()
# Split the fruits by their common separator
strings_of_fruits <- str_split(Fruits,"\"")[[1]]
# using str_split to split the elements of the Fruits string
# into substrings using \" as a separator for the character vectors
EmptyList <- strings_of_fruits[c(FALSE,TRUE)]
# Assign the list back to its original variable
Fruit <- EmptyList
Fruit
```

```
##  [1] "bell pepper"  "bilberry"     "blackberry"   "blood orange" "blueberry"
##  [6] "cantaloupe"   "chili pepper" "cloudberry"   "elderberry"   "lime"
## [11] "lychee"       "mulberry"     "olive"        "salal berry"
```

This string of fruits is formatted as a bunch of character vectors as is the example. We could check the validity of the match with the example string by looking for duplicates and logical testing. If the following logical expression outputs all values of "TRUE" with no extras, then we know we have a perfect match.

```
Fruit == Fruit_example
```

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

Et voila! We now know that we have exact matches for each of the fruits to the fruit_example.

Describing Expressions

Describe in words what each of the following expressions will match. To test them, the list of words below can be used as an example reference.

```
expression_words <- as.character(c("potato", "cheese", "alfalfa",
                        "-", "dystopia", "wonderful",
                        "potato", "racecar", "buttercup",
                        "abba", "tenet", "opera",
                        "ha1a1","zoooooom", "abacas"))
is.character(expression_words)
```

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

Expression 1

- "(.)\1\1"

This expression will not match with any characters because it will resolve to escape the next character in the regular expression. Though it is very close to calling for the same characters 3 times in a row, it is essentially unfinished. We should see the absence of a match in our tests below where "character(0)" displays as the output, indicating that there are no words that match this pattern or regular expression in the values we chose.

```
str_subset(expression_words, "(.)\1\1")
```

```
## character(0)
```

As expected, character(0) appears as the output as though there are no strings that match. We can check if it produces any results with the other Fruit and Fruit_example variables as well.

```r
str_subset(Fruit, "(.)\1\1")
```

```
## character(0)
```

```r
str_subset(Fruit_example, "(.)\1\1")
```

```
## character(0)
```

We can also validate this with another function, str_match, which will extract any matches and display them below.

```r
# validating to search for any match that could be applicable visually
str_match(expression_words, "(.)\1\1")
```

```
##       [,1] [,2]
##  [1,] NA   NA
##  [2,] NA   NA
##  [3,] NA   NA
##  [4,] NA   NA
##  [5,] NA   NA
##  [6,] NA   NA
##  [7,] NA   NA
##  [8,] NA   NA
##  [9,] NA   NA
## [10,] NA   NA
## [11,] NA   NA
## [12,] NA   NA
## [13,] NA   NA
## [14,] NA   NA
## [15,] NA   NA
```

```r
# results for each of the word strings we have is the same, due to the expression itself
```

As you can see, there are no applicable matches.

However, the expression below will match with any identical characters that appear 3 times consecutively in a string. To me, it seems this was intended in the first expression. It can be rewritten as such:

- "(.)\1\1"

Now, we test this expression.

```r
str_subset(expression_words, "(.)\\1\\1")
```

```
## [1] "zoooooom"
```

```r
str_match(expression_words, "(.)\\1\\1")
```

```
##       [,1]  [,2]
##  [1,] NA    NA  
##  [2,] NA    NA  
##  [3,] NA    NA  
##  [4,] NA    NA  
##  [5,] NA    NA  
##  [6,] NA    NA  
##  [7,] NA    NA  
##  [8,] NA    NA  
##  [9,] NA    NA  
## [10,] NA    NA  
## [11,] NA    NA  
## [12,] NA    NA  
## [13,] NA    NA  
## [14,] "ooo" "o" 
## [15,] NA    NA  
```

It correctly displayed the result as "zooooooom" since it is the only word in the values of expression words that contain 3 identical characters written consecutively. It identified them as the "ooo" in the word zooom.

Expression 2

- "(.)(.)\2\1"

This expression will match with any pair of characters that are duplicated in reverse order of the original character pair within the same string.

```r
str_subset(expression_words, "(.)(.)\\2\\1")
```

```
## [1] "abba"      "zooooooom"
```

```r
str_match(expression_words, "(.)(.)\\2\\1")
```

```
##       [,1]   [,2] [,3]
##  [1,] NA     NA   NA  
##  [2,] NA     NA   NA  
##  [3,] NA     NA   NA  
##  [4,] NA     NA   NA  
##  [5,] NA     NA   NA  
##  [6,] NA     NA   NA  
##  [7,] NA     NA   NA  
##  [8,] NA     NA   NA  
##  [9,] NA     NA   NA  
## [10,] "abba" "a"  "b" 
## [11,] NA     NA   NA  
## [12,] NA     NA   NA  
## [13,] NA     NA   NA  
## [14,] "oooo" "o"  "o" 
## [15,] NA     NA   NA  
```

Here again we can see the word zoooooom because it has at least two "o" characters that are the same when written in reverse order. Abba is another good example since a-b written in reverse is b-a.

Expression 3

- "(..)\1"

This expression will match with any two identical characters that get repeated in the same string.

```
str_subset(expression_words, "(..)\\1")
```

```
## [1] "ha1a1"     "zoooooooom"
```

```
str_match(expression_words,"(..)\\1")
```

```
##       [,1]   [,2]
##  [1,] NA     NA
##  [2,] NA     NA
##  [3,] NA     NA
##  [4,] NA     NA
##  [5,] NA     NA
##  [6,] NA     NA
##  [7,] NA     NA
##  [8,] NA     NA
##  [9,] NA     NA
## [10,] NA     NA
## [11,] NA     NA
## [12,] NA     NA
## [13,] "a1a1" "a1"
## [14,] "oooo" "oo"
## [15,] NA     NA
```

It correctly identified the al-al sequence in the string of "halal" as requested. Meanwhile, the constant "zoooooom" makes another appearance due to its o's.

Expression 4

- "(.).\1.\1"

This expression will match with any character specified, followed by any unspecified character, followed by the specified character, followed by any unspecified character.

```
str_subset(expression_words, "(.).\\1.\\1")
```

```
## [1] "zoooooooom" "abacas"
```

```
str_match(expression_words, "(.).\\1.\\1")
```

```
##       [,1]    [,2]
##  [1,] NA      NA
##  [2,] NA      NA
##  [3,] NA      NA
##  [4,] NA      NA
##  [5,] NA      NA
##  [6,] NA      NA
##  [7,] NA      NA
##  [8,] NA      NA
##  [9,] NA      NA
## [10,] NA      NA
## [11,] NA      NA
## [12,] NA      NA
## [13,] NA      NA
## [14,] "ooooo" "o"
## [15,] "abaca" "a"
```

It looks for matches of every other character in strings. Abacas was subset from the string and is a good example since its contains the characters a_a_a in that order. Whereas the string "zoooooom" had its o's satisfy this expression.

Expression 5

- "(.)(.)(.).*\3\2\1"

This expression will match with three identical characters, then either nothing or any number of any characters, with the same three identical characters written in reverse order at the end of the string.

```r
str_subset(expression_words, "(.)(.)(.).*\\3\\2\\1")
```

```
## [1] "racecar"   "zoooooom"
```

```r
str_match(expression_words, "(.)(.)(.).*\\3\\2\\1")
```

```
##       [,1]      [,2] [,3] [,4]
##  [1,] NA        NA   NA   NA
##  [2,] NA        NA   NA   NA
##  [3,] NA        NA   NA   NA
##  [4,] NA        NA   NA   NA
##  [5,] NA        NA   NA   NA
##  [6,] NA        NA   NA   NA
##  [7,] NA        NA   NA   NA
##  [8,] "racecar" "r"  "a"  "c"
##  [9,] NA        NA   NA   NA
## [10,] NA        NA   NA   NA
## [11,] NA        NA   NA   NA
## [12,] NA        NA   NA   NA
## [13,] NA        NA   NA   NA
## [14,] "oooooo" "o"  "o"  "o"
## [15,] NA        NA   NA   NA
```

It searches for identical characters to the first 3 but written in reverse order. The words racecar and zooooom were subset from the string since racecar is a palindrome and the o's in zoom is also identical written forwards and backwards.

Constructing Expressions

Directions

In this case we are to construct regular expressions to match words that: * Start and end with the same character. * Contain a repeated pair of letters (e.g. "church" contains "ch" repeated twice.) * Contain one letter repeated in at least three places (e.g. "eleven" contains three "e"s.) Each will be displayed as a separate task. First, I will duplicate the words variable from *R for Data Science* by Garrett Grolemound and Hadley Wickham.

Recreating Words Example From *R for Data Science* by Garrett Grolemound and Hadley Wickham we have the following string.

```
words_rexample <- '[1] "appropriate" "church"      "condition"   "decide"      "environment"
#>  [6] "london"      "paragraph"   "particular"  "photograph"  "prepare"
#> [11] "pressure"    "remember"    "represent"   "require"     "sense"
#> [16] "therefore"   "understand"  "whether" "doood" "alfalfa'
# Borrowing same chunk for converting this nasty string above
FillerList <- vector()
strings_of_words <- str_split(words_rexample,"\"")[[1]]
FillerList <- strings_of_words[c(FALSE,TRUE)]
words <- FillerList
words
```

```
##  [1] "appropriate" "church"      "condition"   "decide"      "environment"
##  [6] "london"      "paragraph"   "particular"  "photograph"  "prepare"
## [11] "pressure"    "remember"    "represent"   "require"     "sense"
## [16] "therefore"   "understand"  "whether"     "doood"       "alfalfa"
```

Start and end with the same character Using the rules from above, this is an example of a function that returns a word that starts and ends with the same character from the example data from *R for Data Science.*

```
str_subset(words, "^(.)((.*\\1$)|\\1?$)")
```

```
## [1] "remember" "doood"    "alfalfa"
```

It takes the first letter of each string and looks at the last letter to determine if there is a match. If the first and last letter match, then the output is the full string.

Contain a repeated pair of letters Here again, we reference the same example database to find words that have a repeated pair of letters.

```
str_subset(words, "([A-Za-z][A-Za-z]).*\\1")
```

```
##  [1] "appropriate" "church"      "condition"   "decide"      "environment"
##  [6] "london"      "paragraph"   "particular"  "photograph"  "prepare"
## [11] "pressure"    "remember"    "represent"   "require"     "sense"
## [16] "therefore"   "understand"  "whether"     "alfalfa"
```

Importantly, this syntax references the A-Z alphabet we are familiar with using [A-Za-z]. Then, any character pair can be used, it just has to repeat itself in the words. There are quite a few.

Contain one letter repeated in at least three places Lastly, the str_subset function will pull from the word strings any words that contain one character repeated in at least 3 places. Again, we are specifically using the standard A-Z alphabet.

```
str_subset(words, "([a-z]).*\\1.*\\1")
```

```
## [1] "appropriate" "environment" "paragraph"   "remember"    "represent"
## [6] "therefore"   "doood"       "alfalfa"
```

The rule says to look for any character of the standard alphabet ([a-z]) that occurs twice by using (*\1) that is then seen again in the same string, then once again, for a total of three occurrences within one string. If any words satisfy this expression, show the full string, or strings, in a list.
