Data 607 - Assignment 3

Richie R.

Overview

In this assignment, we will go over string manipulation. This includes simple functions as well as regex operations.

Assignment Tasks

Task 1:

Using the 173 majors listed in fivethirtyeight.com's College Majors dataset [https://fivethirtyeight.com/features/the-economic-guide-to-picking-a-college-major/], provide code that identifies the majors that contain either "DATA" or "STATISTICS"

```r
library(magrittr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(stringr)

majors_url <- "https://raw.githubusercontent.com/fivethirtyeight/data/master/college-majors/majors-list
majors_df <- read.csv(majors_url)

head(majors_df)
```

```
##   FOD1P                                 Major                  Major_Category
## 1  1100                   GENERAL AGRICULTURE Agriculture & Natural Resources
## 2  1101 AGRICULTURE PRODUCTION AND MANAGEMENT Agriculture & Natural Resources
## 3  1102                AGRICULTURAL ECONOMICS Agriculture & Natural Resources
## 4  1103                      ANIMAL SCIENCES Agriculture & Natural Resources
## 5  1104                          FOOD SCIENCE Agriculture & Natural Resources
## 6  1105            PLANT SCIENCE AND AGRONOMY Agriculture & Natural Resources
```

Now that we've read in the dataset, we'll need to find all entries which contain either "DATA" or "STATISTICS"

One way to do this is by using the str_detect() function:

```
filtered_majors <- majors_df %>%
  filter(str_detect(Major, "DATA") | str_detect(Major, "STATISTICS"))

filtered_majors
```

```
##   FOD1P                                         Major          Major_Category
## 1  6212 MANAGEMENT INFORMATION SYSTEMS AND STATISTICS                Business
## 2  2101      COMPUTER PROGRAMMING AND DATA PROCESSING Computers & Mathematics
## 3  3702                STATISTICS AND DECISION SCIENCE Computers & Mathematics
```

This seems to work, although my preferred method would be to have a list of entries we would like to check for. By using a list, we can easily expand our search to include any new keywords.

In order to use my preferred method, we'll follow the below steps:

1. Create a list of all the entries we want to search for
2. Initialize an empty dataframe to collect our positive results. This will be done by just filtering the majors_df dataframe to return 0 entries.
3. iterate through each element in the list and add any entries that match to the empty dataframe.

```
keywords_to_check <- list("DATA", "STATISTICS")

keyword_collection_df <- majors_df %>%
  filter(1 == 2)

for (keyword in keywords_to_check){
  keyword_collection_df <- union(
    keyword_collection_df,
    majors_df %>%
      filter(str_detect(Major, keyword))
  )
}

keyword_collection_df
```

```
##   FOD1P                                         Major          Major_Category
## 1  2101      COMPUTER PROGRAMMING AND DATA PROCESSING Computers & Mathematics
## 2  6212 MANAGEMENT INFORMATION SYSTEMS AND STATISTICS                Business
## 3  3702                STATISTICS AND DECISION SCIENCE Computers & Mathematics
```

Let's also try this using regex:

```
regex_pattern <- "DATA|STATISTICS"

majors_df %>%
  filter(str_detect(Major, pattern = regex_pattern))
```

```
##   FOD1P                                           Major          Major_Category
## 1  6212 MANAGEMENT INFORMATION SYSTEMS AND STATISTICS                Business
## 2  2101      COMPUTER PROGRAMMING AND DATA PROCESSING Computers & Mathematics
## 3  3702               STATISTICS AND DECISION SCIENCE Computers & Mathematics
```

Comparing the results of these three methods, it seeems that the order has changed but the number of results has not.

Task 2:

Write code that transforms the data below: [1] "bell pepper" "bilberry" "blackberry" "blood orange" [5] "blueberry" "cantaloupe" "chili pepper" "cloudberry"
[9] "elderberry" "lime" "lychee" "mulberry"
[13] "olive" "salal berry" Into a format like this: c("bell pepper", "bilberry", "blackberry", "blood orange", "blueberry", "cantaloupe", "chili pepper", "cloudberry", "elderberry", "lime", "lychee", "mulberry", "olive", "salal berry")

As I interpret this, it seems that this task is requiring one of the two below:

1. Encode this information as a vector
2. Convert a vector and use cat() to have the terminal display that exact string

The following 2 chunks of code will do each of these two.

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

fruit_vector
```

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

Now we will convert the vector into that exact string. Specifically: c("bell pepper", "bilberry", "blackberry", "blood orange", "blueberry", "cantaloupe", "chili pepper", "cloudberry", "elderberry", "lime", "lychee", "mulberry", "olive", "salal berry")

```
string_prefix <- 'c("'
string_suffix <- '")'
string_sep    <- '", "'

string_output <- str_c(string_prefix,
                       str_flatten(fruit_vector, string_sep),
                       string_suffix)

cat(string_output)
```

```
## c("bell pepper", "bilberry", "blackberry", "blood orange", "blueberry", "cantaloupe", "chili pepper"
```

With input from the week 3 meetup:

To do so this way, we'll perform 2 steps:

1. Create a regex pattern that will match on each fruit and the quotes and then extract this information into a list.
2. With that list, convert it to a vector. As a vector, we will use str_flatten and concatenate the string with the prefix and suffix.

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

string_prefix <- 'c("'
string_suffix <- '")'
string_sep    <- ", "

string_pattern <- '"(.*?)"'

list_of_elements <- str_extract_all(input_string, pattern = string_pattern)

vector_of_elements <- unlist(list_of_elements)

combined_string <- str_flatten(vector_of_elements, string_sep)

output_string <- str_c(string_prefix, combined_string, string_suffix)

cat(output_string)
```

```
## c(""bell pepper", "bilberry", "blackberry", "blood orange", "blueberry", "cantaloupe", "chili pepper
```

Using that 3rd method, we can see that the string is outputted in the format specified.

Task 3:

Describe, in words, what these expressions will match:

1. (.)\1\1

The '(.)' represents any character and the '\1' represents matching the same text as the previous group, which is the '(.)'. Therefore, this would match any 3 of the same character (excluding new line characters). That being said, it may not work as the '\1' is not escaped correctly

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

There are 2 groups here, but they are both '(.)'. The '\2' represents that the 2nd group must be repeated and '\1' represents that the 1st group must be repeated. This essentially means that a matching string is any string of 4 characters where the 1st and 4th character are the same and the 3rd and 4th character are the same. For example, this would match on appa, ette, and eeee.

3. (..)\1

The '(..)' is the first and only group and it represents any 2 characters (except new line). Then, the '\1' represents that the 1st group must be repeated. This would mean that any 2 character pattern that is repeated twice. For example: erer, abab, 1212, and 1111. That being said, it may not work as the '\1' is not escaped correctly

4. "(.).\1.\1"

The only group here is the first group which is '(.)'. After that, there can be any character (except new line) and then the first group repeated twice more. So essentially, a string of 4 characters where the 1st, 3rd, and 4th characters must be the same and the 2nd character can be anything (except a new line). For example: 1211, abaa, oeoo, kˆkk.

5. "(.)(.)(.).*\3\2\1"

There are 3 groups of '(.)' which mean any 3 characters. following any 3 characters, there is a '.*' which represents matching on any number of any characters (except new lines). Finally, the back references at the end are essentially calling the groups from the beginning in reverse order. Plainly, this is looking for any string that begins with any 3 characters and ends with the same 3 characters reversed. For example: 123000000000321, abc[whole script to a move]abc, aaa[another whole move]aaa, etc

Task 4:

Construct regular expressions to match words that:

1. Start and end with the same character.

For this we will use \b, \1, and [A-Za-z].

```r
t4_1_pattern <- "\\b([A-Za-z])[A-Za-z]*\\1\\b"

t4_1_test <- c("meriam", "health", "meriam is in perfect health")

str_view(t4_1_test, pattern = t4_1_pattern)
```

```
## [1] | <meriam>
## [2] | <health>
## [3] | <meriam> is in perfect <health>
```

2. Contain a repeated pair of letters (e.g. "church" contains "ch" repeated twice.)

```r
t4_2_pattern <- "\\b[A-Za-z]*([A-Za-z][A-Za-z])[A-Za-z]*\\1[A-Za-z]*\\b"

t4_2_test <- c("church", "mississippi",
               "there are many churches in mississippi")

str_view(t4_2_test, t4_2_pattern)
```

```
## [1] | <church>
## [2] | <mississippi>
## [3] | there are many <churches> in <mississippi>
```

3. Contain one letter repeated in at least three places (e.g. "eleven" contains three "e"s.)

```r
t4_3_pattern <- "\\b[A-Za-z]*([A-Za-z])[A-Za-z]*\\1[A-Za-z]*\\1[A-Za-z]*\\b"

t4_3_test <- c("narrator", "antenna", "believe", "parallel", "purple",
               "the narrator believes that antenna must be parallel")

str_view(t4_3_test, t4_3_pattern)
```

```
## [1] | <narrator>
## [2] | <antenna>
## [3] | <believe>
## [4] | <parallel>
## [6] | the <narrator> <believes> that <antenna> must be <parallel>
```

Conclusion

Although I've had exposure to regex in the past, I have never felt very confident in my ability to write or understand it. After running through these operations I feel much more comfortable writing and feel capable to read regex.