Data 607 - Assignment 3 - R String Manipulation

William Jasmine

9/14/2019

Problem 1

Description

Using the 173 majors listed in fivethirtyeight.com's College Majors dataset, provide code that identifies the majors that contain either "DATA" or "STATISTICS".

Solution

First, the majors data is imported from the `majors-list.csv` file and stored as an R dataframe called `df`.

```
df <- read.csv("majors-list.csv", header=TRUE)
glimpse(df)
```

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

The `glimpse()` function reveals that the `df` dataframe has three columns, and 174 rows (each of which pertaining to a single major).

In the cell below, two steps take place:

- A regular expression string (`regex`) is created to match all major names with the desired conditions. The regex used is `.*STATISTICS.*|.*DATA.*`, which checks for instances of the words "STATISTICS" or "DATA" that can be preceded or followed by any other characters (using `.*`).
- The `regex` string is used in conjunction with the `dplyr::filter()` and `grepl()` functions to return all matching rows in the `df` dataframe.

```
regex = ".*STATISTICS.*|.*DATA.*"
df %>% dplyr::filter(grepl(regex, Major))
```

```
##   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
```

The results of the code above reveal that there are three majors that contain the terms "DATA" or "STATISTICS": COMPUTER PROGRAMMING AND DATA PROCESSING, STATISTICS AND DESICION SCIENCE, and MANAGEMENT INFORMATION SYSTEMS AND STATISTICS.

Problem 2

Description

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", "
```

Solution

First, the code chunk below initializes the data as a string and stores it in the `fruits_str` variable:

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

The following chain of commands completes a number of steps to manipulate the string into the desired format:

1. The `gregexpr()` function uses the regex string `"([a-z]| )*"` to strip out all the locations and lengths of the fruit names from `fruit_str`. The regex does this by matching all double quote enclosed collections of letter characters and white space.
2. The `regmatches()` function takes the output from the step above and produces a character vector of all the matches when applying the regex to `fruit_str`.
3. Since the matches from the regex will include double quote characters (`"`), the `gsub()` function is then used to remove them.

The output of these three steps is shown below and is stored as `fruit_vec`:

```
fruits_vec <-
  regmatches(fruits_str, gregexpr('"([a-z]| )*"', fruits_str))[[1]] %>%
    gsub(pattern = '"', replacement = "")
fruits_vec
```

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

This looks to be exactly the output required, which is confirmed in the cell below:

```
fruits_vec == c("bell pepper", "bilberry", "blackberry", "blood orange", "blueberry", "cantaloupe", "chi
```

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

Problem 3

Description

Describe, in words, what these expressions will match:

a. `(.)\1\1`
b. `"(.)(.)\\2\\1"`
c. `(..)\1`
d. `"(.).\\1.\\1"`
e. `"(.)(.)(.).*\\3\\2\\1"`

Solution (Part a)

Regex: `(.)\1\1`

Explanation:

This regex will match any string that contains a single character followed by `\1\1`. In order for the `\1` to have the backreference effect in R, a double backslash (`\\`) needs to be used so that the first backslash can escape the second. Because this hasn't happened, the `\1` is matched directly.

Tests:

```
# define regex string
regex = "(.)\1\1"
# define test cases
test_case1 <- "a\1\1" # match
test_case2 <- "a\1\1a" # match
test_case3 <- "a\1a\1" # no match
test_case4 <- "\1\1" # no match
test_case5 <- "\1\1a" # no match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5)

# create vector of expected results
expected <- c(TRUE, TRUE, FALSE, FALSE, FALSE)

# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
results == expected
```

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

The results of the tests shown above prove that the explanation or the regex matches its intended results.

Solution (Part b)

Regex: `"(.)(.)\\2\\1"`

Explanation:

This regex will match any string that contains a four character, double quote enclosed palindrome (same from left to right as right to left). The two `(.)` represent the first and second capturing groups, which can

be any single character. The `\\2\\1` then means that the following two characters must be from the second and first capturing group in that order (the first `\` characters in this case are properly escaped and will have the desired backreference effect, unlike in part a). Finally, this collection of four characters must be enclosed in double quotes in order for the regex to match.

Tests:

```
# define regex string
regex = '"(.)(.)\\2\\1"'
# define test cases
test_case1 <- '"abba"' # match
test_case2 <- 's"abba"o' # match
test_case3 <- '"ab|ba"' # no match
test_case4 <- '"sabbat"' # no match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4)

# create vector of expected results
expected <- c(TRUE, TRUE, FALSE, FALSE)

# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
results == expected
```

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

The results of the tests shown above prove that the explanation or the regex matches its intended results.

Solution (Part c)

Regex: `(..)\1`

Explanation:

This regex will match any string that contains two single characters `(..)` followed by a `\1`. Once again, the `\1` does not do a backreference due to the fact that the initial `\` is not escaped by another `\`.

Tests:

```
# define regex string
regex = '(..)\1'
# define test cases
test_case1 <- "ab\1" # match
test_case2 <- 'ab\1a' # match
test_case3 <- 'a\1' # no match
test_case4 <- 'abc\1a' # match
test_case5 <- 'abba' # no match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5)

# create vector of expected results
expected <- c(TRUE, TRUE, FALSE, TRUE, FALSE)
```

```r
# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
results == expected
```

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

The results of the tests shown above prove that the explanation or the regex matches its intended results.

Solution (Part d)

Regex: `"(.).\\1.\\1"`

Explanation:

This regex will match any string that contains a double quote enclosed substring of 5 characters in which the 1st, 3rd, and 5th characters of that substring are the same. The `(.)` corresponds to the first capture group of the regex, which must be repeated twice more at two and four characters later (according to the placement of the backreferences `\\1`). Any character can be used in the 2nd and 4th spot of the 5 character substring (due to the `.`), and that substring must be be enclosed in double quotes `"`.

Tests:

```r
# define regex string
regex = '"(.).\\1.\\1"'
# define test cases
test_case1 <- '"abaca"' # match
test_case2 <- 's"abaca"t' # match
test_case3 <- '"abacd' # no match
test_case4 <- 'abaca' # no match
test_case5 <- '"abdac' # no match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5)

# create vector of expected results
expected <- c(TRUE, TRUE, FALSE, FALSE, FALSE)

# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
results == expected
```

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

The results of the tests shown above prove that the explanation or the regex matches its intended results.

Solution (Part e)

Regex: `"(.)(.)(.).*\\3\\2\\1"`

Explanation:

This regex will match any string that contains a double quote enclosed substring in which the first three characters of that substring are reversed and included as the last three characters of that same substring.

The three `(.)` define three capture groups, each comprising a single character that should immediately follow the first double quote character. The `\\3\\2\\1"` then looks to make sure that the third, second, and first capture group are found before the closing double quote character at the end of the substring (in that order). The `.*` in the middle means that any other characters can exist between these sets of characters in the substring.

Tests:

```
# define regex string
regex = '"(.)(.)(.).*\\3\\2\\1"'
# define test cases
test_case1 <- '"abccba"' # match
test_case2 <- '"abc|99cba"' # match
test_case3 <- 's"abccba"t' # match
test_case4 <- '"abcabc"' # no match
test_case5 <- 'abccba' # no match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5)

# create vector of expected results
expected <- c(TRUE, TRUE, TRUE, FALSE, FALSE)

# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
results == expected
```

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

The results of the tests shown above prove that the explanation or the regex matches its intended results.

Problem 4

Description

Construct regular expressions to match words that:

a. Start and end with the same character.
b. Contain a repeated pair of letters (e.g. “church” contains “ch” repeated twice.)
c. Contain one letter repeated in at least three places (e.g. “eleven” contains three “e”s.)

Solution (Part a)

Regex Used: `^(.).*\\1$|^(.)$`

Explanation:

- The first `^(.)` captures the first character of the string and saves it as the first capturing group.
- The `\\1$` then ensures that the same first capturing group exists at the end of the string.
- The `.*` in the middle allows there to be any other characters between the first and last ones.
- Finally, the `|^(.)$` also allows for any single characters to match, since they technically start and end with the same character.

Tests:

```r
# define regex string
regex = "^(.).*\\1$|^(.)$"
# define test cases
test_case1 <- "a"  # match
test_case2 <- "ab" # no match
test_case3 <- "aa" # match
test_case4 <- "afgkgdfshb" # no match
test_case5 <- "agjgfslbga" # match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5)

# create vector of expected results
expected <- c(TRUE, FALSE, TRUE, FALSE, TRUE)

# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
expected == results
```

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

The results of the tests above prove that the regex string is working correctly.

Solution (Part b)

Regex Used: `([a-z][a-z]).*\1.*`

Explanation:

- The `([a-z][a-z])` matches all adjacent pairs of letters as the first capturing group.
- The `\\1` checks to see if any of the adjacent letter pairs stored in the first capturing group repeat later in the string.
- The two `.*` entries are to ensure that there can be any number of characters between and after the repeated letter pairs.

Tests:

```r
# define regex string
regex = "([a-z][a-z]).*\\1.*"
# define test cases
test_case1 <- "ab"  # no match
test_case2 <- "abba" # no match
test_case3 <- "abab" # match
test_case4 <- "church" # match
test_case5 <- "churhc" # no match
test_case6 <- "churchy" # match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5, test_case6)

# create vector of expected results
expected <- c(FALSE, FALSE, TRUE, TRUE, FALSE, TRUE)
```

```r
# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure tests align with expected results (all outputs should be TRUE)
results == expected
```

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

The results of the tests above prove that the regex string is working correctly.

Solution (Part c)

Regex Used: `([a-z]).*\1.*\1.*`

Explanation:

- The `([a-z])` matches all single letters as the first capturing group.
- The first `\\1` checks to see if any of single letters stored in the first capturing group repeat for a first time later in the string.
- The second `\\1` checks to see if any of single letters stored in the first capturing group repeat for a second time later in the string.
- The three `.*` entries are to ensure that there can be any number of characters between the repeated single characters.

Tests:

```r
# define regex string
regex = "([a-z]).*\\1.*\\1.*"
# define test cases
test_case1 <- "a" # no match
test_case2 <- "aaa" # match
test_case3 <- "aba" # no match
test_case4 <- "eleven" # match
test_case5 <- "elevan" # no match
test_case6 <- "elevene" # match

# create character vector with all test cases
tests <- c(test_case1, test_case2, test_case3, test_case4, test_case5, test_case6)

# create vector of expected results
expected <- c(FALSE, TRUE, FALSE, TRUE, FALSE, TRUE)

# use regex to determine outcome of test cases
results <- grepl(regex, tests)

# ensure that all tests align with expected results
results == expected
```

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

The results of the tests above prove that the regex string is working correctly.